A2000
A500

Amiga Basic
Svensk

Commodore

A2000
A500

Amiga Basic
Svensk

Commodore

Amiga Basic är utvecklat av Microsoft Corporation och bygger på Microsoft BASIC.

Kopieringsrätt (Copyright):
Kopieringsrätten © till denna manual tillkommer helt Commodore-Amiga Inc. och Microsoft Corporation, 1985, alla rättigheter reserverade. Hela eller delar av denna manual får inte kopieras, fotostatkopieras, reproduceras, översättas eller lagras på något elektroniskt media eller på något sätt som kan läsas av en maskin utan skriftligt tillstånd från Commodore-Amiga Inc.

Kopieringsrätten © till denna mjukvara tillkommer helt Microsoft Corporation, 1985, alla rättigheter reserverade. Distributionen och försäljningen av denna produkt är avsedd för originalförsäljaren endast. Lagliga användare av detta program är härmed licensierade till att endast läsa detta program från sitt medium till minnet hos en dator i syfte att exekvera programmet. Duplicering, kopiering, försäljning eller någon annan form av mångfaldigande eller distribution av denna produkt är ett brott mot lagen.

Förbehåll:
Detta program levereras "som det är" utan någon garanti, uttryckt eller underförstådd, inkluderat, men inte begränsat till de underförstådda sälj-garantierna och lämplighet för ett speciellt ändamål. Alla risker i samband med programmets funktion tages av användaren. Om programmet visar sig vara defekt så får användaren själv (och inte utvecklaren eller Commodore Inc. eller dess försäljare) ta hand om hela kostnaden för nödvändig service, reperation eller rättningar. Commodore-Amiga garanterar inte resultaten av användningen av programmet gällande korrekthet, noggranhet, tillförlitlighet eller annat; användaren litar på programmet och dess resultat helt på egen risk. I inget fall är Commodore-Amiga Inc. ansvariga för direkta, indirekta, tillfälliga eller därav följande skador beroende på någon defekt i programmet även om Commodore-Amiga Inc. har blivit informerade om möjligheterna av sådana skador. Vissa lagar tillåter ej undantaganden från eller begränsningar av de underförstådda säljgarantierna eller ansvar för tillfälliga eller därav följande fel så begränsningarna eller undantagandena ovan gäller ej alltid.

Microsoft är ett varumärke registrerat hos Microsoft Corporation.
Amiga är ett varumärke registrerat hos Commodore-Amiga Inc.
Macintosh är ett varumärke registrerat av Apple Computers.
IBM-PC är ett varumärke registrerat av IBM Inc.

Originalets titel: Amiga Basic

Översättning och bearbetning: Gustaf Arrhenius, LEXICON Produktion AB

ISBN 91-7944-041-X

CBM P/1:

CBM P/N: 380798-05

Innehåll

Kapitel 1: Amiga Basic: en introduktion 1-1

Kapitel 2: Kom igång med Amiga Basic! 2-1

Kapitel 3: Hur använder Amiga Basic 3-1

Kapitel 4: Redigera och felsöka i program 4-1

Kapitel 5: Att arbeta med filer och logiska enheter 5-1

Kapitel 6: Avancerad programmering 6-1

Kapitel 7: Skapa animerade bilder 7-1

Kapitel 8: Amiga Basic: referensdel 8-1

Appendix A-1

Index I-1

1 Amiga Basic: en introduktion

Vem använder Basic? Många olika kategorier av människor använder sig av Basic: en del är professionella programmerare, andra är nybörjare som aldrig förut hållit på med programmering.

Den största andelen Basic-användare utgörs antagligen av de som skriver program för eget bruk. Dessa kanske skriver egna program för att de tycker om den intellektuella utmaningen, eller för att de behöver ett program till ett mycket speciellt användningsområde, så speciellt att det inte finns något program att köpa som kan lösa problemet. Studerande utgör en annan stor grupp Basic-användare, de använder Basic för att lära sig programmera och som hjälp i skolarbetet.

Alla Basic-användare har en sak gemensam: de använder Basic för att det är ett universellt språk för mikrodatorer. Det är lätt att lära, finns tillgängligt på de flesta mindre datorer och är mycket standardiserat. Det är ett mångsidigt språk; det används för att skriva allt från avancerade vetenskapliga och tekniska program till enkla program i utbildningssyfte och inte minst för att skriva spelprogram.

Amiga Basic

Amiga Basic har alla kommandon, instruktioner och funktioner som finns i standard-Basic. Kärnan i Amiga Basic bygger på Microsoft Basic, det mest sålda programspråket till mikrodatorer. Men Amiga Basic erbjuder en mängd extrafunktioner: man kan skriva väl strukturerade program och uttnyttja Amigans fantastiska bild- och ljudmöjligheter. Man kan använda sig av rullgardinsmenyer, pekare, fönsterteknik mm.

Om den här handboken

Den här handboken beskriver Amiga Basic-tolken (interpretatorn). Man behöver vissa förkunskaper för att kunna förstå förklaringarna i handboken: man måste ha kunskaper motsvarande det som behandlas i **Amiga användarhandbok**, dvs vara bekant med menyer, textredigering,

handhavande av musen mm.

Handbokens första sju kapitel beskriver:

* allmänt om hur man kommer igång
* handhavande av Amiga Basic på Amigasystemet
* hur man använder Amiga Basic-tolken
* hur man skriver in och ändrar i program
* felsökning
* hur man hanterar filer och extrautrustning (logiska enheter)
* hur man använder objekteditorn
* avancerad programmering.

Kapitel 8 (som utgör ungefär hälften av denna handbok) är en referensdel till Amiga Basic. Där finns alla kommandon, instruktioner, funktioner och operatorer listade i bokstavsordning och där redogörs för deras fullständiga syntax.

Amiga Basic:s egenskaper

Amiga Basic är skriven i assembler och upptar därför lite primärminne (80 Kb). Kärnan i Amiga Basic har testats i praktiska tillämpningar i närmare tre år. De flesta program skrivna i Microsoft Basic kan tolkas (dvs köras) av Amiga Basic. Amiga Basic förbättras successivt, det kommer nya versioner med en utökad repertoar av kommandon, instruktioner och funktioner för att hålla Amiga Basic aktuell med det senaste inom datatekniken.

Amiga Basic har ett antal kraftfulla egenskaper som man har stor nytta av:

Block-satser

Med IF...THEN...ELSE-satser kan man utföra val i program. Man kan ha flera satser på en eller flera rader efter "THEN".

Underprogram

Underprogram kan ha egna lokala variabler. Med hjälp av dessa underprogram kan man bygga upp ett bibliotek av Basic-rutiner som man kan använda till flera olika program. Man behöver inte bekymra sig

om att ha olika variabelnamn i huvudprogram och underprogram.

SHARED-instruktionen

Med SHARED-instruktionen kan man definiera variabler till att bli delade, dvs både huvud- och underprogram kan använda sig av och påverka deras värden.

Heltal

Man kan använda sig av både 16- och 32-bitars heltal.

Flyttal

Man kan använda sig av både 32- och 64-bitars flyttal.

Radnummer

Man behöver inte använda radnummer.

Alfanumeriska radadresser

Alfanumeriska radadresser kan användas för att skapa adresser som beskriver vad en programsekvens gör. Alfanumeriska radadresser gör program lättare att läsa, förstå och underhålla.

Sekventiella filer och direktåtkomstfiler

Man kan använda sig av både sekventiella filer och direktåtkomstfiler (random access). Sekventiella filer är enklare att skapa, medan direktåtkomstfiler är mer flexibla och snabbare på att lokalisera data.

Hantering av I/O

Med särskilda instruktioner för filhantering kan man också hantera in- och utdata från skärmen, tangentbordet, skrivare, RS232-porten och parallellporten. Man skickar data till en skrivare på samma sätt som man skriver data på en skiva.

Hantering av Amigans speciella egenskaper

Amiga Basic har instruktioner och funktioner som är speciellt avsedda för att hantera Amigans utmärkta färg-, grafik-, animerings- och ljudmöjligheter:

* Synkroniserad fyrstämmig musik skapas med SOUND- och WAVE-instruktionerna.

* Med SAY- och TRANSLATE- instruktionerna kan man skapa syntetiska tal.

* Mycket snabb hantering av skärmbilder sker med GET- och PUT-instruktionerna.

* Grafik skapas t ex med instruktionerna LINE, CIRCLE, PAINT, AREA, AREAFILL.

* Möjligheter att skapa animerade bilder finns genom OBJECT-instruktionen, objekteditorn och COLLISION-funktionen.

* Möjlighet att anropa maskinspråksrutiner finns med LIBRARY- och DECLARE-instruktionerna.

* Fönsterhantering (flera fönster på en gång!) utförs av instruktionerna SCREEN och WINDOW.

* Man kan skapa och hantera rullgardinsmenyer med MENU-instruktionen och MENU()-funktionen.

Alla dessa instruktioner och funktioner beskrivs i detalj i kapitel 8, "Amiga Basic: referensdel". Några av dem beskrivs summariskt nedan.

SOUND och WAVE

Man kan framställa ljud av hög kvalitet med Amiga Basic. SOUND-instruktionen framställer en ton med den frekvens, längd och volym som man önskar. Tonen kan också få en av fyra egendefinierade

"röster". Med WAVE-instruktionen kan man skapa olika vågformer som man sedan kan tilldela en röst. Med SOUND och WAVE kan man framställa en mängd olika ljud; allt från en stråkkvartett till en visslad låt är möjligt att få fram!

LINE och CIRCLE

LINE och CIRCLE är två mångsidiga instruktioner som används då man skapar grafik.

LINE drar en linje mellan två punkter. Punkterna kan vara absoluta eller relaterade till någon annan punkt. Med tillvalet B så ritar LINE en rektangel, med tillvalet BF så ritas en rektangel och fylls med en färg.

CIRCLE ritar en cirkel, cirkelbåge eller en ellips med angiven centrumpunkt och radie. Man kan även ange vilken färg cirkeln skall ha, och vilket höjd/bredd-förhållande som skall användas. Det sista tillvalet gör att man kan rita ellipser i olika storlekar och former.

GET, PUT och SCROLL

GET läser ett område av skärmen och lagrar det. På så sätt kan man enkelt lagra bilder i minnet. Med PUT kan man rita upp bilden på skärmen igen. Med SCROLL kan man definiera ett område och ange åt vilket håll och med vilken hastighet det skall röra sig.

Objekteditorn

Till Amiga Basic finns ett hjälpprogram, den s k objekteditorn, med vilket man kan skapa bilder som man kan använda i sina egna program. I kapitel 7, "Skapa animerade bilder", beskrivs objekteditorn i detalj.

Hur kan man lära sig mer om Basic?

Den här handboken innehåller inte så många övningar i Basic. Om du är nybörjare i programmering så föreslår vi att du läser någon av följande böcker:

* **BASIC Handboken,** D Lien
* **VIC-64 Basicboken,** D Hergert
* **Lär dig programmera, del 1,** P Cornes

2 Kom igång med Amiga Basic!

För att kunna använda Amiga Basic behöver man:

* En uppkopplad Amiga-dator.
* Amiga Extras-skivan.

Man bör också ha två säkerhetskopior av Amiga Extras-skivan, så kopiera över Extras-skivans innehåll till två tomma skivor.

Starta Amigan:

* Slå på strömmen till systemenheten och bildskärmen.
* När datorn ber om Kickstart-skivan, så sätter man in den i skivenheten.
* När Workbench-skivan efterfrågas, tag ur Kickstart-skivan och sätt i Workbench-skivan.
* När Workbench-skivans symbol kommer upp på skärmen, ta ur Workbench-skivan och sätt i Amiga Extras-skivan.
* Öppna Amiga Extras-skivan.
* Öppna Amiga Basic.

Nu framträder Amiga Basic:s karakteristiska skärmutseende.

Anmärkning: I den här handbokens övningar och exempel förutsätts att Amiga Basic-skärmen använder Workbenchens originalfärger, nämligen:

* Blå bakgrund
* Vit förgrund
* Orange
* Svart

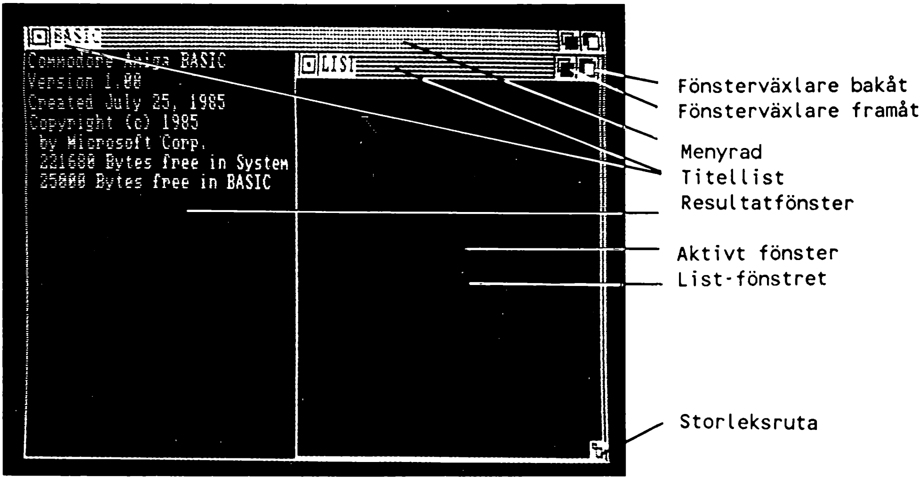
På skärmen ser man List-fönstret. I List-fönstret syns textmarkören som

en lodrät orange linje. Man kan nu skriva in ett nytt program eller läsa in ett befintligt program från en skiva. Notera att titellisten i List-fönstret framträder distinkt medan titellisten i Basic-fönstret är skuggad. Det är för att markera att List-fönstret är aktuellt (valt) i motsats till Basic-fönstret.

I resultatfönstret (just nu Basic-fönstret) visas utdata från ett program (text, siffror, bilder etc). Här kan man också arbeta interaktivt, dvs ge Amiga Basic-kommandon och direkt få svar.

Om man vill arbeta i det interaktiva läget så måste man först göra resultatfönstret aktuellt genom att klicka musens valknapp samtidigt som man pekar på fönstret. Notera att Amiga Basic svarar med Ok-prompten.

För att se menytitlarna som finns på menyraden, tryck ner musens menyknapp.



En praktisk övning

Tid för övningen: Femton minuter

För att visa innehållet på Extras-skivan i resultatfönstret:.

* Välj resultatfönstret.

* När Ok-prompten kommer upp, skriv:

```
files <RETUR>
```

Nu kommer Extras-skivans filnamn och bibliotek upp i resultatfönstret.

Om fönstret blir fullt så rullar namnen uppåt för att ge rum åt fler namn i slutet av listan. För att stanna rullningen tryck höger Amiga-tangent och S-tangenten; för att återstarta rullningen, tryck någon tangent, vilken som helst.

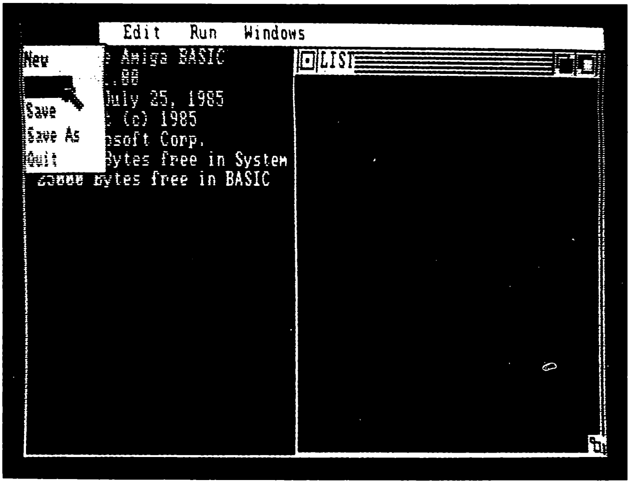
Hämta Picture

Vi skall nu läsa in programmet Picture som finns i BasicDemos-lådan. Picture är ett demonstrationsprogram som följer med Extras-skivan. Det är skrivet i Amiga Basic.

* Tryck ner och håll nere musens menyknapp.

* Peka på Project-menyns titel.

* Välj "Open".



* Peka på namnrutan och klicka musens valknapp.

* Skriv:

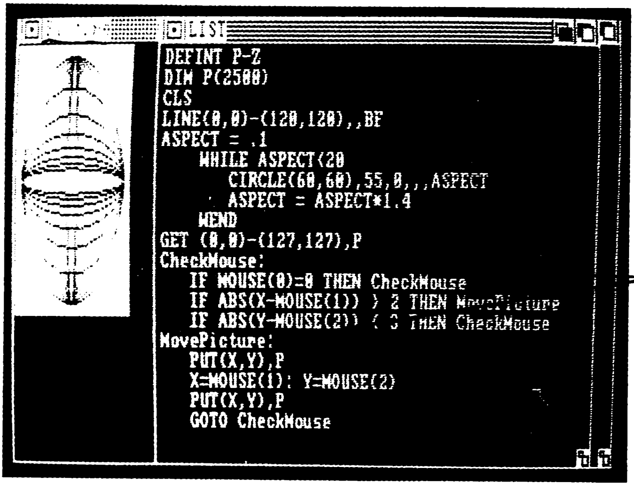
```
basicdemos/picture
```

* Klicka Ok-rutan eller tryck <RETUR>.

Anmärkning: Mer information om fil- och biblioteksnamn finns i *Amigados User's manual.*

En listning av Picture-programmet visas i List-fönstret. Namnet på resultatfönstret ändras från "BASIC" till "basicdemos/picture".

Vid en listning av ett Basic-program väntar man sig kanske att finna radnummer i början av varje rad. Så är inte fallet i Amiga Basic, där radnummer är valfritt. För att kunna referera till en rad ger man den ett radnummer eller en radadress. Picture-programmet har inga radnummer men två radadresser: "CheckMouse" och "MovePicture". Adresser behövs om flödet i ett program skall brytas med hopp till olika satser.



Radadresser

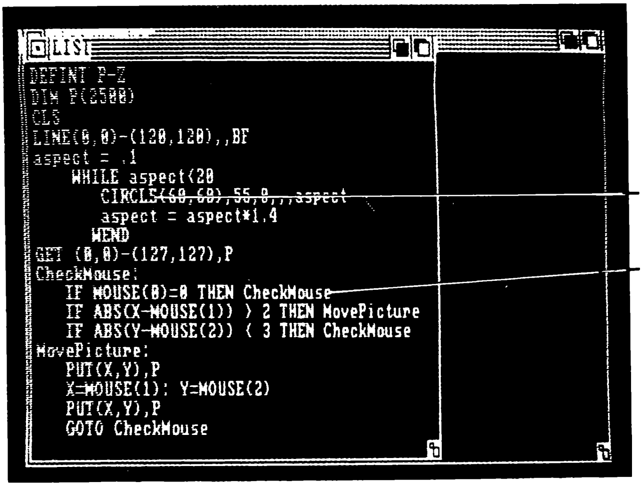
* Välj resultatfönstret.

* Skriv:

```
list CheckMouse <RETUR>
```

Nu visas raden med CheckMouse-adressen och alla efterföljande rader. Vill man ändra något i CheckMouse-rutinen måste man åter göra List-fönstret aktuellt genom att klicka på det.

Reserverade ord: Det är mycket lätt att läsa programlistningar då de reserverade Basic-orden automatiskt konverteras till versaler. Konverteringen sker då man lämnar programraden.



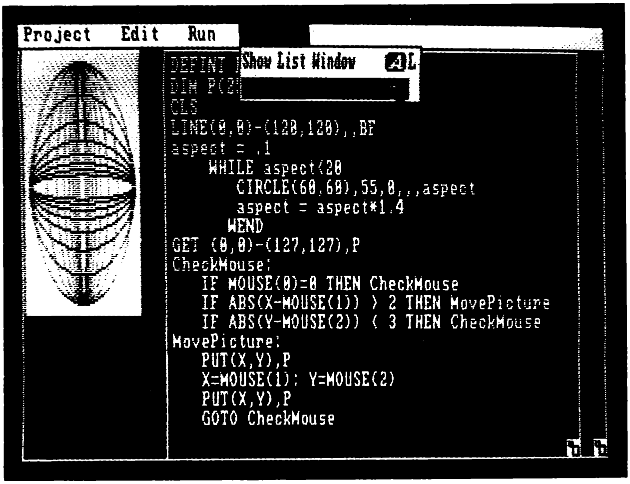
Amiga Basic:s reserverade ord görs om till versaler.

Andra ord förblir som de matades in.

Vad gör Pictureprogrammet?

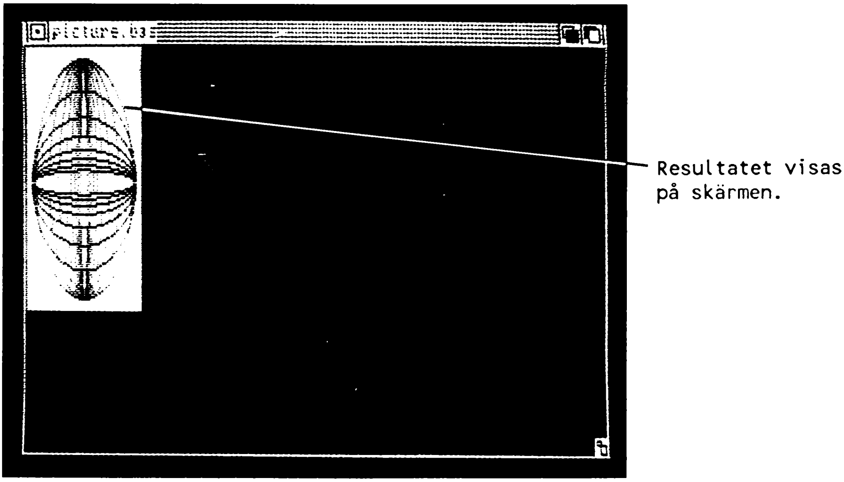
Starta programmet genom att göra så här:

* För att öppna resultatfönstret över List-fönstret välj "Show Output" från Windows-menyn.



* Välj "Start" från Run-menyn.

En sfär ritas i resultatfönstret. Bilden kan flyttas genom att man klickar på olika ställen i resultatfönstret. Pröva!



Stanna programmet

Programmet kommer att fortsätta att arbeta ända tills man avbryter det:

* Välj "Stop" från Run-menyn.

* Välj "Show List" från Windows-menyn.

List-fönstret öppnas igen. Om man vill ändra i programmet så måste man klicka List-fönstret för att göra det aktuellt.

Flytta textmarkören i List-fönstret

För att flytta textmarkören rad för rad använder man sig av piltangenterna nere till höger på tangentbordet. Pil upp och pil ned flyttar markören en rad uppåt respektive en rad nedåt.

Vänster- och högerpil flyttar markören ett tecken åt vänster respektive ett tecken åt höger.

För att flytta markören en sida i taget, tryck SHIFT- tangenten och pil upp

eller pil ned.

Anmärkning: När man i den här handboken ser två tangenter förenade med bindestreck så betyder det att man skall trycka ner och hålla nere den första tangenten samtidigt som man trycker på den andra tangenten.

Exempel:

SHIFT-Pil Upp betyder att man skall trycka ner och hålla nere SHIFT-tangenten samtidigt som man trycker ner pil upp.

För att flytta ned markören framåt sida för sida i programtexten, tryck SHIFT-Pil Ned. För att flytta markören uppåt sida för sida tryck SHIFT-Pil Upp.

Några andra sätt att flytta markören:

ALT-Pil Upp	Flyttar markören till programmets första rad
ALT-Pil Ned	Flyttar markören till programmets sista rad
ALT-Pil Höger	Flyttar markören till slutet på raden
ALT-Pil Vänster	Flyttar markören till början på raden
SHIFT-Pil Höger	Flyttar markören 3/4 av fönstrets bredd åt höger
SHIFT-Pil Vänster	Flyttar markören 3/4 av fönstrets bredd åt vänster

Vill du veta mer om programmet Picture, se Appendix G: "Övningsprogrammet Picture.bas". Där går vi igenom programmet rad för rad.

Att redigera ett program

Att redigera ett Amiga Basic-program är ungefär som att redigera en text med en ordbehandlare. Man matar in programtexten i List-fönstret. I Edit-menyn finns diverse hjälpmedel för att ändra i texten.

Man skriver in ny text genom att:

* Peka på det ställe där den nya texten skall skrivas in.

* Klicka valknappen så att textmarkören flyttas till pekarens position.

* Skriva in den nya texten.

Man tar bort tecken till vänster om markören genom att trycka BACKSPACE-tangenten.

För att välja ut ett ord, dvs måla det, peka på ordet och dubbelklicka valknappen.

För att måla ett helt textstycke gör man så här:

* Markera början av stycket genom att peka på det och klicka valknappen.

* Peka på slutet av stycket och SHIFT-klicka valknappen (dvs tryck ner SHIFT-tangenten och valknappen samtidigt).

Man kan klippa, kopiera och klistra ord och stycken precis som i en vanlig ordbehandlare.

Om man vill se hela programlistningen på en gång så måste man förstora List-fönstret:

* Peka på List-fönstrets titellist och tryck och håll ned valknappen.

* Flytta fönstret till vänster genom att föra musen åt vänster.

* Släpp upp valknappen när fönstret befinner sig längst till vänster.

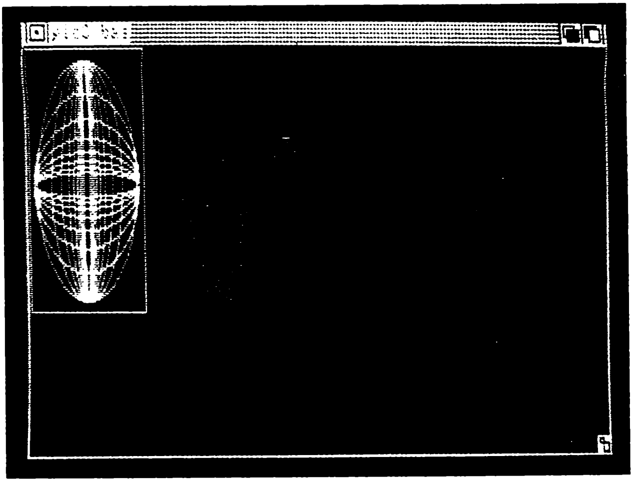
* Peka på storleksrutan (finns i fönstrets nedre högra hörn).

* Tryck och håll ned valknappen.

* Förstora fönstret genom att flytta pekaren nedåt och åt höger.

* När du tycker att fönstret har en lämplig storlek, släpp då upp valknappen.

Vi skall nu pröva att redigera Picture och samtidigt lära oss lite om de grafiska funktioner som finns i Amiga Basic. Var inte rädd för att göra förändringar i programtexten då det finns en exakt kopia av Picture på Extras-skivan kallad Picture2. Vi skall pröva att förändra programmet så att det ger följande bild på skärmen:

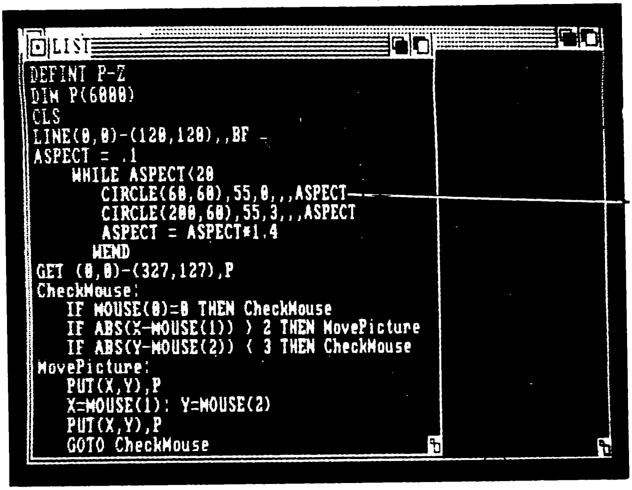


Lägga till en programrad

Vi startar med att lägga till den rad som ritar upp den andra sfären:

* Leta rätt på programraden:

```
CIRCLE(60,60),55,0,,,aspect
```



Programraden som ritar den första sfären

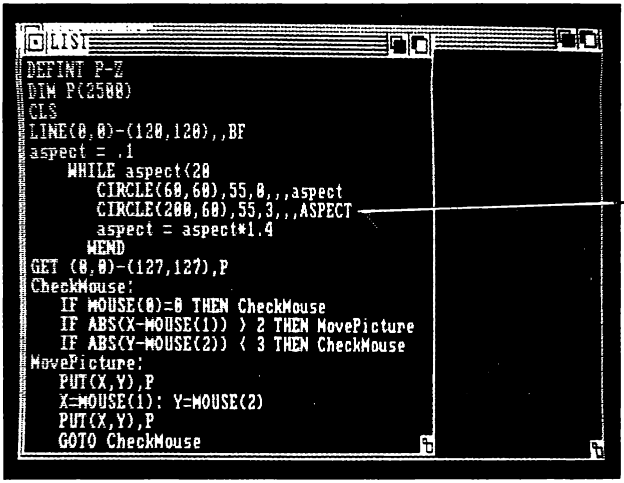
* Peka på slutet av raden och klicka valknappen så att textmarkören flyttas dit.

* Tryck <RETUR> för att få en tom rad.

Notera att editorn automatiskt gör indrag så att nästa rad skrivs i samma kolumn som den föregående.

* Skriv:

```
CIRCLE(200,60),55,3,,,aspect
```



Skriv in den här programraden för att rita upp den andra sfären

Den nya CIRCLE-satsen ritar upp en ellips med centrum i punkten 200,60. Radien är 55 och höjd/bredd-förhållandet varierar med variabeln *aspect*. 3 anger att färgen orange skall användas (0 i den första CIRCLE-satsen anger vit färg). Varje gång WHILE-slingan utförs ritar CIRCLE en ellips med nytt höjd/bredd-förhållande *(aspect)*. Samtliga ellipser bildar sfären.

* Kör nu programmet genom att välja "Start" från RUN-menyn.

Rätta fel

Det kan lätt hända att man skriver fel i ett program. När programmet sedan körs kommer Amiga Basic att stanna när den försöker utföra den

felaktiga satsen. Ett felfönster som anger vilken typ av fel det är öppnas på skärmen. List-fönstret öppnas och i programlistningen är den felaktiga raden inringad av en orange rektangel. Man kan direkt korrigera felet och starta programmet igen.

Ersätta en programrad

I den nya versionen av programmet så visas två sfärer men bara den ena kan förflyttas. Vi skall ändra programmet så att båda sfärerna kan flyttas:

* Om programmet fortfarande körs avbryt det.

* Välj "Show List" från Windows-menyn.

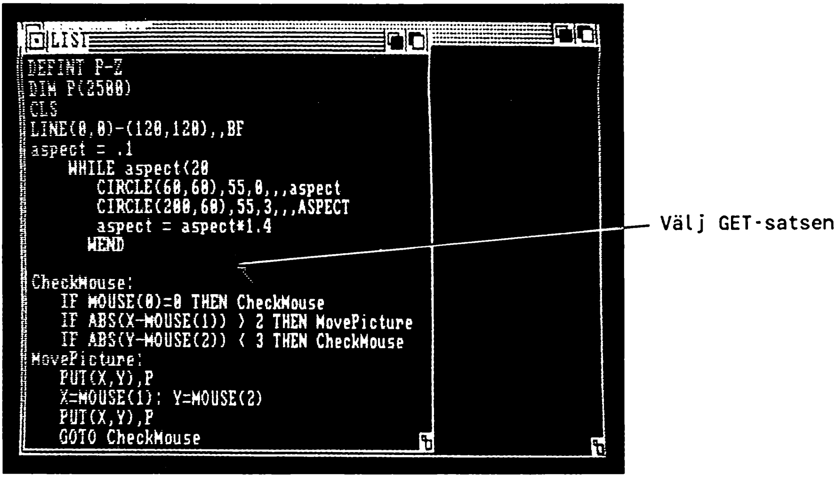
Observera att kommandot "Show List" inte ändrar den nya positionen på List-fönstret.

* Peka längst till vänster på GET-satsen.

* Tryck och håll ned valknappen.

* Flytta pekaren till slutet av GET-satsen så att den oranga markeringen dras över hela GET-satsen.

* Släpp upp valknappen.



* Välj "Cut" från Edit-menyn.

Den ursprungliga raden har tagits bort och ersatts av en blankrad.

* Skriv in följande på den tomma raden:

```
GET(0,0)-(327,127),p
```

Den nya GET-satsen ökar bildarean som flyttas när valknappen klickas.

Vi måste ändra DIM-satsen så att en matris på 6000 element definieras istället för den gamla på 2500 element:

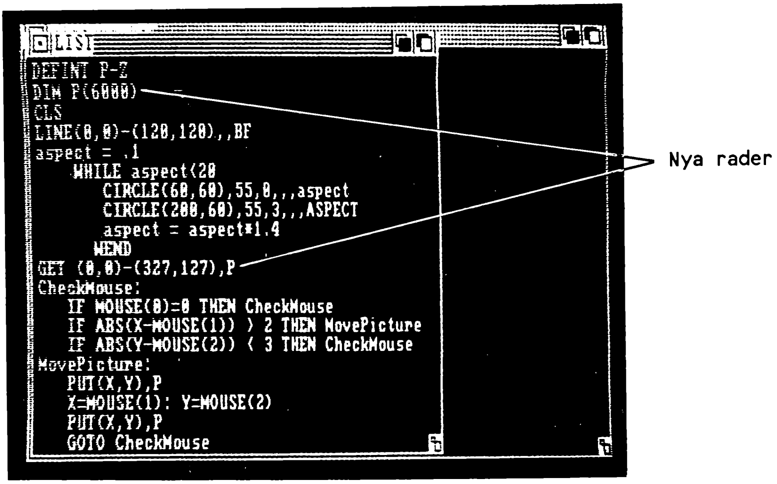
* Flytta textmarkören till DIM-satsen.

* Måla talet "2500".

* Tag bort det med hjälp av "Cut" från Edit-menyn.

* Skriv in talet "6000" inom parenteserna så att den nya raden får följande utseende:

```
DIM p(6000)
```

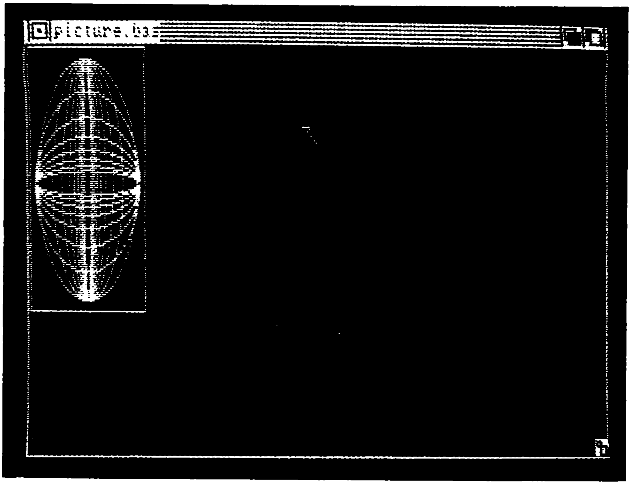


* Välj "Start" från Run-menyn för att köra den nya programversionen.

Pröva att klicka valknappen när du pekar på olika ställen på skärmen. Nu rör sig båda sfärerna!

Ändra färg

Låt oss pröva att ändra färg på den första sfären så att den blir vit på blå bakgrund:

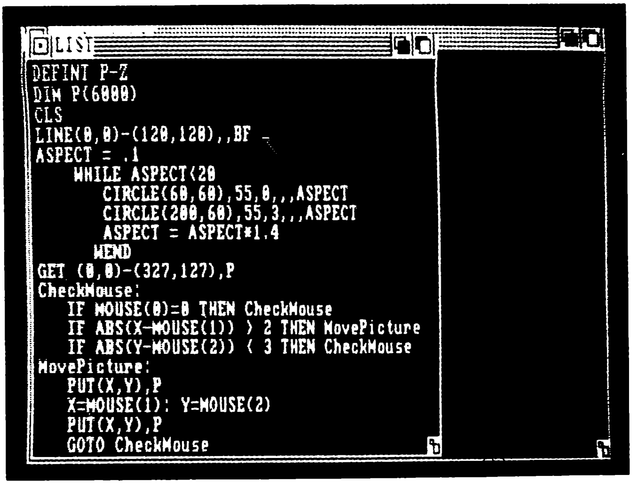


* Om programmet fortfarande är igång, avbryt det med "Stop" från Run-menyn.

* Öppna List-fönstret.

* Leta reda på LINE-satsen.

* Peka på slutet av satsen och klicka på valknappen så att textmarkören hamnar direkt efter "BF".



* Tryck BACKSPACE-tangenten så att "F" i "BF" tas bort.

Nu kommer rektangeln som omger första sfären att bli blå istället för vit.

* Leta rätt på raden:

```
CIRCLE(60,60),55,0,,,aspect
```

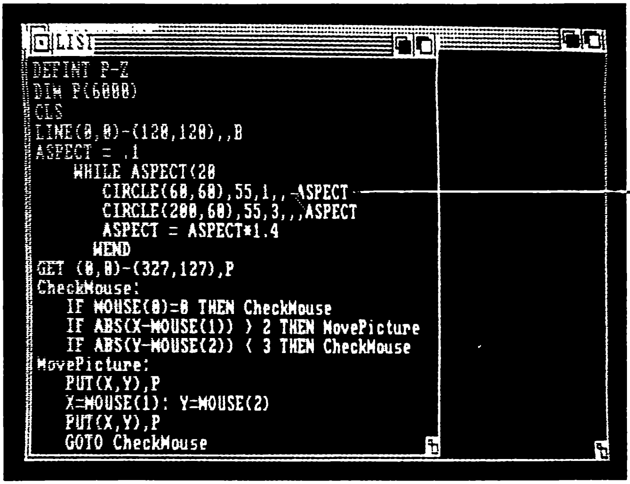
* Placera textmarkören just efter 0:an (den ensamma!).

* Tryck BACKSPACE en gång för att ta bort nollan.

* Skriv:

```
1
```

för att få färg nummer 1.



Nu kommer ellipsen att ritas i vitt istället för i blått.

* Kör programmet.

Programkörning rad för rad

För att vi skall få större kännedom om programmet Picture skall vi utnyttja en ofta använd felsökningsmetod: att köra en rad i taget.

* Avbryt programmet om det fortfarande körs.

* Klicka resultatfönstret så att det blir aktivt.

Lägg märke till Ok-prompten.

* Skriv:

```
end <RETUR>
```

* Välj "Step" från Run-menyn.

Nu utförs programmets första rad, sedan stannar programkörningen tillfälligt.

* Välj "Show List" från Windows-menyn så att List-fönstret öppnas.

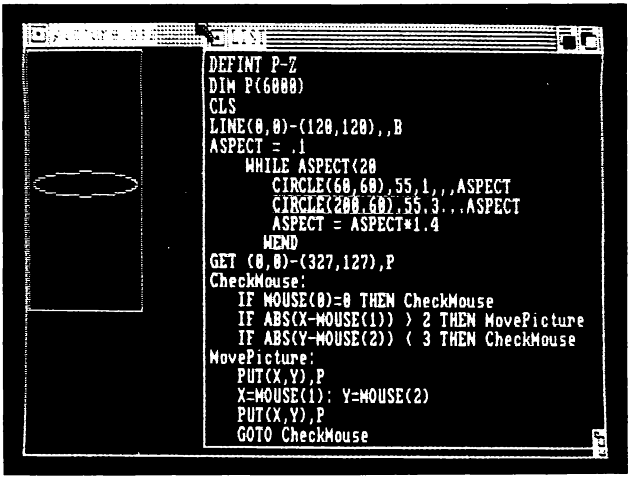
* Flytta List-fönstret till högra sidan av skärmen genom att dra i titellisten.

Den sats som senast utförts är innesluten i en orange rektangel. Resultat-fönstret är aktivt, så att de tecken du matar in från tangentbordet hamnar där.

* Välj "Step" igen (eller tryck höger Amiga-T).

Nästa programrad utförs men då den inte innehåller några utdata till resultatfönstret så händer det inte så mycket.

Fortsätt att välja "Step". När programmet kommer fram till den del som ritar ellipserna, observera då det sätt som sfärerna ritas på. Varje genomlöpning av WHILE-slingan ritar en ellips med nytt höjd/bredd-förhållande *(aspect)* vilka så småningom bygger upp sfärerna.



* När några ellipser blivit uppritade, pröva att skriva:

```
print aspect <RETUR>
```

Det värde som *aspect* har för tillfället skrivs ut i resultatfönstret.

Trots att vi egentligen inte utförde någon riktig felsökning i Picture så illustrerar övningen ändå ett typiskt sätt att felsöka i program.

Man kan skriva in Amiga Basic-satser i resultatfönstret och direkt få dem utförda. Det kallas att arbeta *interaktivt.* När en sats ges interaktivt så utförs den direkt och eventuella utdata visas på en gång. Se "Olika arbetslägen" i kapitel 3 för mer information om Amiga Basic:s interaktiva arbetssätt.

* Fortsätt att köra rad för rad i Picture. Kontrollera gärna olika variablers värden.

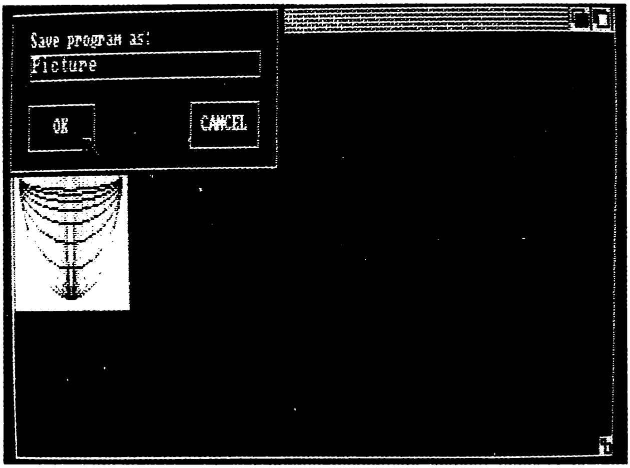
* Om du vill sluta med rad-för-rad körningen och istället köra programmet på vanligt sätt, välj "Continue" från Run-menyn.

Spara programmet

Man använder kommandot "Save As" från Project-menyn då man vill spara ett nytt eller modifierat program på skiva. När ett program är sparat på skiva så finns det kvar även efter det att strömmen brutits så att man vid ett senare tillfälle kan ladda och köra det igen.

* Välj "Save As" från Project-menyn.

Följande frågefönster kommer upp:



Som förvalt värde antar editorn att man vill spara ett modifierat program under dess gamla namn.

Du kan ändra programnamnet om du vill men enklast är att klicka Ok-rutan så att programmet blir sparat under sitt gamla namn.

Nu finns det två versioner av Picture på skivan: En kopia av ursprungsversionen av Picture kallad Picture2 och vår modifierade version av Picture. Om man hade velat kunde man ha ändrat namnet på Picture när man lagrade det, till t ex "MittProgram". I så fall så skulle Picture fortfarande finnas kvar i originalversionen.

Lämna Amiga Basic och återgå till Workbenchen

* Välj "Quit" från Project-menyn.

Du är nu tillbaka på Workbenchen. I den här lektionen har vi hunnit lära oss mycket om Amiga Basic. Vi har lärt oss hur man:

* Läser in ett program från skiva.

* Redigerar program i List-fönstret.

* Arbetar med satser och funktioner.

* Sparar en programfil.

I nästa kapitel kommer vi att gå igenom mer om Amiga Basic. Några av sakerna kommer du att känna igen men vi kommer också att gå igenom mycket nytt.

Man kan aldrig skada vare sig datorn eller Amiga Basic genom att utföra någon sats, funktion eller kommando eller genom att peka med musen. Så tveka aldrig att experimentera själv!

Läs in ett program

För att läsa in ett program som finns på skivan i skivstationen till minnet, skriv:

```
LOAD "filnamn"
```

Redigera ett inläst program

För att ändra i ett program som finns i minnet eller för att skriva in ett nytt program, skriv:

```
LIST [<radadress>]
```

LIST startar fullskärmseditorn. Om man anger en radadress så kommer raden med den angivna adressen högst upp i List-fönstret följd av efterkommande programrader.

Köra ett program som finns i minnet

För att starta ett program som finns inläst i minnet, skriv:

```
RUN
```

För att avbryta programmet, tryck CTRL-C.

För att felsöka i programmet så kan man utnyttja det interaktiva arbetssättet.

Exempel:

```
FOR I=1 TO 20: PRINT A(I): NEXT I
```

För att återuppta programkörningen, skriv:

```
CONT
```

Lämna Amiga Basic

För att lämna Amiga Basic och återvända till Workbenchen, skriv:

```
SYSTEM
```

Spara ett program som finns i minnet

Om programmet i minnet inte är lagrat när du ger kommandot SYSTEM så öppnas ett frågefönster på skärmen med texten:

```
Current program is not saved
Do you want to save it before proceeding?
```

Klicka YES-rutan om du vill spara programmet, NO-rutan om du inte vill spara programmets nya version och CANCEL-rutan om du vill återvända till Amiga Basic.

3 Hur man använder Amiga Basic

Detta kapitel ger grunderna i hur man hanterar Amiga Basic, bl a:

* Hur man läser in och sparar filer.
* Amiga Basic:s olika arbetslägen.
* Skärmen.
* Menyraden och snabbval till menykommandona.
* Övrigt.

Grundläggande hantering

Följande avsnitt beskriver hur man startar och avslutar Amiga Basic och hur man läser in och sparar program.

Starta Amiga Basic

Det finns två sätt att starta Amiga Basic:

1. Direkt

   * Öppna Extras-skivans fönster genom att att dubbelklicka Extras-symbolen.

   * Öppna Amiga Basic genom att dubbelklicka Amiga Basic-symbolen.

2. Indirekt

   * Öppna Extras-skivan.

   * Öppna en låda innehållande Amiga Basic-program.

   * Öppna ett program genom att dubbelklicka en programsymbol.

Amiga Basic kommer att automatiskt öppnas när man öppnar ett Amiga

Basic-program. Programmet kommer dessutom att köras direkt.

Avsluta Amiga Basic och återvända till Workbenchen

Det finns tre sätt att avsluta Amiga Basic:

1. Välj "Quit" från Project-menyn (som finns på menyraden).

2. Skriv:

   ```
   system &lt;RETUR&gt;
   ```

   i resultatfönstret.

3. Skriv:

   ```
   SYSTEM
   ```

   som en instruktion i ett program.

Inläsning av program

För att köra ett program så måste det finnas i minnet. Det finns många sätt att läsa in ett program från skiva till minnet:

1. När man befinner sig på Workbenchen så kan man dubbelklicka en Amiga Basic programsymbol. Då läses Amiga Basic och programmet in i minnet. Programmet körs sedan automatiskt.

2. Om Amiga Basic redan är inläst i minnet så kan man välja "Open" från Project-menyn. Då öppnas ett frågefönster där man anger vilket program som skall läsas in. Klicka namnrutan, skriv in programnamnet och klicka Ok-rutan.

3. Om Amiga Basic är inläst kan man i resultatfönstret skriva något av kommandona LOAD eller RUN. Se kapitel 8 för fullständig syntax.

4. Instruktionen CHAIN i ett program läser in ett nytt program.

Spara program på skiva

Det finns två sätt att spara ett nytt eller modifierat program på skiva:

1. Välj "Save" från Project-menyn.

2. Skriv:

```
save <RETUR>
```

i resultatfönstret.

Amiga Basic sparar program i binär form om man inte anger någon annan form. För att spara ett program i textformat eller i skyddad form så måste man ge extra instruktioner med SAVE-kommandot. Se kapitel 8 för fullständig syntax.

Olika arbetslägen

När man öppnat Amiga Basic kommer resultatfönstret (under namnet "BASIC") och List-fönstret upp på skärmen. Man kan då välja mellan tre olika arbetslägen:

* Interaktiva läget.

* Redigeringsläget.

* Körläget.

Interaktiva arbetsläget

När man arbetar interaktivt med Amiga Basic så lagras inte kommandon och satser i minnet utan de utförs direkt efter inmatning i resultatfönstret. Resultaten av de aritmetiska eller logiska operationerna visas direkt i resultatfönstret och sparas även i minnet så att man kan använda dem senare.

Det interaktiva arbetsläget är mycket användbart då man felsöker i program och då man vill utföra enklare beräkningar där det inte behövs ett helt program för att lösa uträkningarna.

För att kunna arbeta i det interaktiva läget måste man göra resultatfönstret aktivt genom att klicka någonstans inuti det.

Redigeringsläget

Man är i redigeringsläget när man arbetar i List-fönstret.

Körläget

När ett program körs så är man i körläget. Man kan då varken använda sig av det interaktiva läget eller redigeringsläget utan att avbryta programmet.

Amiga Basic-skärmen

Det finns tre olika områden på Amiga Basic-skärmen:

* List-fönstret.
* Resultatfönstret.
* Menyraden.

List- och resultatfönstret har följande punkter gemensamma:

* För att välja fönstret, klicka någonstans inuti det.
* För att ändra storleken på fönstret, använd storleksrutan i nedre högra hörnet på fönstret.
* För att flytta fram fönstret till främsta positionen på skärmen, använd fönsterväxlare fram (finns i övre högra hörnet).
* För att flytta fönstret till en position bakom övriga fönster använd fönsterväxlare bak (finns i övre högra hörnet).
* För att stänga fönstret, klicka på stängrutan i övre vänstra hörnet.

* För att flytta fönstret, peka på dess titellist, tryck ner och håll nere valknappen, flytta fönstret och släpp slutligen upp valknappen. Man kan också flytta resultatfönstret om man först ändrar dess storlek.

Menyraden har flera utmärkande egenskaper:

* För att visa menyraden, välj List- eller resultatfönstret och tryck och håll ned menyknappen.

* För att visa en meny, peka på önskad menytitel.

* För att välja ett kommando från en meny, peka på kommandot så att det blir markerat och släpp slutligen upp menyknappen.

I de följande avsnitten skall vi gå igenom ytterligare egenskaper hos List- och resultatfönstret samt menyraden.

Resultatfönstret

Resultatfönstret används för att arbeta interaktivt och för att visa utdata från program.

För att välja resultatfönstret:

* Klicka någonstans inuti fönstret, eller

* välj "Show Output" från Windows-menyn och klicka sedan i resultatfönstret som öppnats på skärmen.

I resultatfönstret kan man:

* Ge kommandon och satser interaktivt så att de utförs direkt när man avslutat dem med <RETUR>. Eventuella resultat visas direkt i fönstret.

* Använda BACKSPACE-tangenten för att ta bort skrivfel.

* Trycka CTRL-C eller Amiga-punkt för att avbryta programkörning.

List-fönstret används till att mata in, ändra och titta på ett program. Man kan också använda det till att se vilken sats som just utförs, vilket är användbart vid felsökning. List-fönstret blir automatiskt aktivt när man öppnar Amiga Basic.

För att välja List-fönstret:

* Klicka i fönstret, eller

* välj "Show List" från Windows-menyn och klicka sedan inuti List-fönstret som öppnas på skärmen.

List-fönstret öppnas även då ett program stannar på grund av programfel.

Observera! Om ett program blivit sparat i skyddad form (med SAVE-kommandot i resultatfönstret) kan man inte öppna ett List-fönster för filen. Skyddade filer kan varken listas eller redigeras.

I List-fönstret kan man:

* Titta på raderna i ett program och bläddra upp och ner i programmet med hjälp av piltangenterna (och SHIFT- och ALT-tangenterna tillsammans med piltangenterna).

* Mata in och ändra i ett program med hjälp av Edit-menyn och musen. Se även "Redigera program" i kapitel 4.

Menyraden och snabbval

Det finns fyra menyer på menyraden:

* Project-menyn.

* Edit-menyn.

* Run-menyn.

* Windows-menyn.

Man kan inte alltid komma åt alla dessa menyer. En menytitel visas ibland mindre distinkt (skuggad) för att markera att den inte går att använda för tillfället. Även ett enskilt kommando i en meny kan vara skuggat för att markera att man inte kan använda det för tillfället.

Bredvid en del av menykommandona står en tangentkombination, t ex Amiga-X för "Cut" på Edit-menyn. Detta betyder att man kan trycka den angivna tangentkombinationen istället för att välja kommandot från menyn med hjälp av musen. Alla sådana **snabbval** av menykommandon använder sig av höger Amiga-tangent.

Project-menyn

Project-menyn innehåller fem kommandon som används för att hantera programfiler. Det finns inga snabbval för kommandona på Project-menyn.

New

New gör att man kan skriva in ett nytt program. "New" raderar List- och resultatfönstret. Minnet raderas från tidigare program för att ge plats åt det nya programmet. NEW-instruktionen i program fungerar på samma sätt.

Open

Med "Open" anger man att man vill läsa in ett program som finns på skiva. För att se namnen på de program som finns på skivan, skriv FILES i resultatfönstret. När man väljer "Open" så öppnas ett frågefönster som frågar efter namnet på programmet som skall läsas in. Klicka namnrutan, skriv in det önskade programnamnet och klicka Ok-rutan så läses programmet in.

Save

Med "Save" sparar man programmet på skiva under sitt senaste namn. "Save" sparar nyinmatade program i binär form och modifierade program i den form de hade när de lästes in.

Save As

"Save As" gör samma sak som "Save" förutom att man kan ändra namn på programfilen om man vill.

För att spara ett program i textformat eller i skyddad form så måste man skriva SAVE direkt i resultatfönstret. Se "Handhavande av filer " i kapitel 5 för förklaring av olika filformat. Se SAVE i kapitel 8 för kommandots fullständiga syntax.

Quit

"Quit" gör att man hoppar ur Amiga Basic och återvänder till Workbenchen. "Quit" fungerar på samma sätt som SYSTEM-kommandot i resultatfönstret och som SYSTEM-instruktionen i program.

Edit-menyn

I Edit-menyn finns det tre kommandon som man använder när man redigerar program i List-fönstret. Alla tre Edit-kommandona har snabbval.

Cut

"Cut" tar bort den text som man målat (med hjälp av musen) och placerar det på klippbordet. Snabbval: Amiga-X.

Copy

"Copy" placerar en kopia av den målade texten på klippbordet. Inget tas bort. Snabbval: Amiga-C.

Paste

"Paste" (klistra) ersätter målad text med klippbordets innehåll. Om ingen text är målad så placeras klippbordets innehåll till höger om textmarkören. Snabbval: Amiga-P.

Run-menyn

Run-menyn innehåller sex kommandon som används för programkörning. Snabbval finns för fyra av dessa kommandon.

Start

"Start" kör programmet som finns i minnet. Man kan också skriva RUN i resultatfönstret. Snabbval: Amiga-R.

Stop

"Stop" avbryter det program som körs. En STOP-instruktion i ett program fungerar på samma sätt. Snabbval: Amiga-C eller CTRL-C.

Suspend

"Suspend" avbryter programmet tills någon valfri tangent trycks ner. Programkörningen återupptas då från den rad den stannade på. "Suspend" finns tillgänglig så fort ett program körs. Snabbval: Amiga-S eller CTRL-S.

Continue

"Continue" startar ett avbrutet program ("Stop" eller "Suspend"). Kommandot CONT i resultatfönstret utför samma sak. "Continue" finns tillgängligt när ett program verkligen har blivit avbrutet. Om inget program är avbrutet eller om man ändrat i programmet efter uppehållet så öppnas ett felfönster när man väljer "Continue" vilket meddelar: "Can't Continue".

Trace On/Off

"Trace" är ett växelkommando som växlar mellan "Trace On" (spårning på) och "Trace Off" (spårning av). Om List-fönstret är synligt och "Trace" är "On" så markeras den senaste utförda satsen i programmet med en orange rektangel. "Trace Off" tar bort markeringen. TRON-instruktionen i ett program fungerar likadant. "Trace" gör att det blir enkelt att hitta den sats i programmet som är orsaken till att programmet avbröts.

Step

Med "Step" körs en sats i taget. Programmet stannar efter varje sats. Om List-fönstret är öppet och synligt markeras den sats som senast utförts genom att den är omgiven av en orange rektangel. Snabbval: Amiga-T.

Windows-menyn

Windows-menyn har två kommandon som används för att öppna fönster på Amiga Basic-skärmen. Ett snabbval finns.

Show List

Show List öppnar och gör List-fönstret aktuellt. Om List-fönstret redan är öppet men täckt av resultatfönstret så flyttas List-fönstret fram. Snabbval: Amiga-L.

Show Output

Show Output öppnar resultatfönstret. Om List-fönstret är öppet placeras det bakom resultatfönstret. Om man vill arbeta i resultatfönstret måste man göra det aktuellt genom att klicka inuti det.

4 Redigera och felsöka i program

Detta kapitel beskriver hur man skriver in och felsöker i program.

Redigera program

List-fönstret öppnas på skärmen när man startar Amiga Basic. Man kan då mata in ett nytt program eller redigera ett redan existerande program. Använd Edit-menyns kommandon för att klippa, klistra och kopiera text. List-fönstret kan verka för smalt för långa satser men när man kommer till fönstrets högra position så rullar texten åt vänster så att textmarkören alltid finns i den synliga delen av List-fönstret. För att komma tillbaka till vänstermarginalen, tryck ALT-vänsterpil. Man kan också göra List-fönstret större: flytta det först till vänster och använd sedan storleksrutan.

Lathund för redigering

Att redigera ett program är ungefär som att arbeta med en vanlig text i en ordbehandlare.

Nedan sammanfattar vi hur man redigerar ett program.

Skriva in och redigera text

* Skriv in text direkt eller klistra in den från klippbordet med "Paste" från Edit-menyn.

* Ta bort text med BACKSPACE-tangenten eller med "Cut" från Edit-menyn.

* Avsluta varje rad med <RETUR>. Man kan också skapa tomma rader i ett program genom att trycka <RETUR> upprepade gånger. Tomma rader påverkar inte programmets funktion.

* Man skapar indrag med hjälp av TAB-tangenten som förflyttar sig tre tecken i taget. När man trycker <RETUR> så hoppar textmarkören ner en rad till den kolumn där föregående rad började. Det betyder att om föregåede rad är indragen så blir den nya raden indragen lika långt. Det kostar inget extra minnesutrymme att ha indrag.

* Man kan skriva in reserverade ord med både versaler och gemena men de visas alltid som versaler.

* Man kan ha variabler med upp till fyrtio (40) signifikanta tecken. En variabel är av typen enkel precision om man inte hänger på speciella typdefinierade tecken eller utför en DEFINT-, DEFLNG-, DEFSNG-, DEFDBL- eller DEFSTR- instruktion. De speciella tecken man använder för att definiera olika variabeltyper är "$" för strängar, "!" för enkel precision, "#" för dubbel precision, "%" för heltal och "&" för långa heltal. Man kan skriva variabelnamn med både versaler och gemena men Amiga Basic skiljer inte på dem. Exempelvis så refererar alpha, Alpha och ALPHA till samma variabel.

* Man *kan* börja en programrad med ett nummer men det är inte nödvändigt.

Välja text

* Välj ett textstycke genom att måla det med hjälp av musen.

* Snabbaste sättet att måla en rad är att peka längst till vänster på raden, hålla ned valknappen och sedan flytta pekaren en rad nedåt.

* Om man målar ända ned till textfönstrets slut så kommer texten att rulla uppåt samtidigt som den blir målad.

* Måla enskilda ord genom att peka på önskat ord och sedan dubbelklicka musens valknapp.

* Ett alternativt sätt att måla ett stycke är att klicka i början av stycket och SHIFT-klicka i slutet av stycket.

Rullning av text

* Om man skriver in rader i slutet av List-fönstret så kommer texten att rulla upp en rad i taget så att de nya raderna får plats.

* När man når List-fönstrets högra sida så rullar texten automatiskt till vänster.

* Använd de fyra piltangenterna för att flytta ett tecken åt höger eller vänster och en rad upp eller ner.

* Om man är vid List-fönstrets högra kant och trycker högerpil så kommer programtexten att rulla 3/4 av fönstrets bredd åt vänster. Om texten inte kan rulla längre till vänster, dvs markören är så långt till höger som det är möjligt, så indikerar en ljudsignal att man inte kan flytta markören längre. De andra piltangenterna fungerar på samma sätt.

* Om man trycker ner SHIFT-tangenten tillsammans med en piltangent så flyttas markören 3/4 av fönstrets bredd i pilens riktning, dvs texten rullar åt pilens motsatta håll. Om markören inte kan flyttas längre så indikeras det med en ljudsignal.

* För att komma till början av programmet, tryck ALT-Pil Upp.

* För att komma till slutet av programmet, tryck ALT-Pil Ned.

* För att flytta markören till slutet av raden, tryck ALT-Pil Höger.

* För att flytta markören till början av raden, tryck ALT-Pil Vänster.

Öppna List-fönstret vid en utvald rad

För att öppna List-fönstret och få en önskad rad att visas högst upp i fönstret, skriv LIST i resultatfönstret följt av den önskade radens adress eller radnummer.

```
LIST MovePicture <RETUR>
```

öppnar List-fönstret och visar raden MovePicture högst upp i fönstret.

Felsöka i program

Detta avsnitt behandlar Amiga Basic:s fyra hjälpmedel vid felsökning:

* Felmeddelanden.

* TRON-instruktionen.

* "Step"-kommandot.

* "Suspend"-kommandot.

Felmeddelanden

Om ett program innehåller ett fel så kommer programkörningen att avbrytas och ett felfönster med ett felmeddelande öppnas. Den felaktiga raden markeras i List-fönstret. I appendix B, "Felkoder och felmeddelanden", finns en komplett lista över alla felmeddelanden och felkoder. Här finns också förslag på vad felen kan bero på och hur de kan åtgärdas.

TRON-kommandot

Man aktiverar TRACE genom att ge kommandot TRON (TRace ON) i resultatfönstret, utföra TRON-instruktionen i ett program eller genom att välja "Trace On" från Run-menyn.

Om List-fönstret är öppet och synligt och TRACE är aktiverat så kommer den senast utförda satsen i det program som körts att vara inramad av en orange rektangel.

Man stänger av TRACE genom att välja "Trace Off" från Run-menyn, ge kommandot TROFF i resultatfönstret eller genom att utföra TROFF-instruktionen i ett program.

Om man med säkerhet kan hänföra ett programfel till en viss sektion av ett program så är det mest praktiskt att använda TRON-instruktionen just före detta programavsnitt.

Step

"Step" kör den sats som står på tur hos programmet i minnet. Om programmet har körts och blivit avbrutet så kör "Step"-satsen efter senast utförda sats. Om det är mer än en sats per rad så utför "Step" varje sats för sig. Man väljer "Step" från Run-menyn eller trycker höger Amiga-T.

Om List-fönstret är framme så ramar "Step in" den senast utförda satsen med en orange rektangel.

Med kommandot PRINT i resultatfönstret kan man kontrollera olika variablers värden mellan "Step"-kommandona.

För att avsluta "Step", skriv:

```
end <RETUR>
```

i resultatfönstret.

Om ON BREAK används i ett program så kan "Step" inte användas tillsammans med det programmet. För mer information, se BREAK i kapitel 8.

Suspend

För att få en paus i programkörningen kan man välja "Suspend" från Run-menyn eller trycka höger Amiga-S. Programkörningen återupptas då man trycker någon annan tangent eller väljer "Continue" från Run-menyn. "Suspend" finns tillgängligt så fort ett program körs.

Continue

För att återuppta en avbruten programkörning, skriv CONT i resultatfönstret eller välj "Continue" från Run-menyn.

Glöm inte bort att klippbordets innehåll ersätts varje gång man använder "Cut"- och "Copy"-kommandona. "Paste" däremot påverkar inte klippbordets innehåll så man kan klistra samma text på flera ställen.

Ibland vill man ta bort ett stycke text utan att skriva över klippbordets innehåll. Det kan man göra genom att måla textstycket som skall tas bort och sedan trycka BACKSPACE-tangenten. Det är också ett sätt att undvika felmeddelandet "Out of heap space" som kan uppkomma då man tar bort stora textstycken.

Resultatfönstrets användning vid felsökning

När ett program blivit avbrutet kan man använda resultatfönstret för att få information om det avbrutna programmet. Om t ex ett program producerar ett felmeddelande och felet uppstår i en slinga så kan man ta reda på hur många gånger slingan har utförts och studera variablernas värden. Man får fram informationen genom att använda kommandot PRINT i resultatfönstret. För fullständig syntax se referensdelen kapitel 8.

Ett annat sätt att utnyttja resultatfönstret i felsökningssyfte är att ändra olika variablers värden med LET-kommandot. Se LET i kapitel 8.

5 Att arbeta med filer och logiska enheter

I detta kapitel går vi igenom hur man hanterar data, filer och logiska enheter i Amiga Basic. Dessutom går vi igenom hur man kan överföra filer mellan ordbehandlare och Amiga Basic.

I/O till olika logiska enheter

Amiga Basic har förenklat syntaxen för de satser som sköter om hanteringen av in- och utdata till olika enheter. Syntaxen har generaliserats så att man använder samma typ av satser oavsett om det gäller sk logiska enheter eller filer på skiva. Följande logiska enheter kan hanteras:

SCRN: Filer kan öppnas till bildskärmen, dvs filens innehåll skickas som utdata till det aktiva resultatfönstret.

KYBD: Filer kan öppnas för att ta emot indata från tangentbordet.

LPT1: Filer kan öppnas till "LPT1:" för att skicka data till skrivarporten. Alla data som skrivs till en fil öppnad som "LPT1:" skickas till en skrivare. Om man lägger till "BIN" ("LPT1:BIN") så skickar Amiga Basic ut filen i binärt format. Då kommer tabulatortecken inte att konverteras till mellanslag och <RETUR> kommer inte att framtvingas när skrivarens bredd är överskriden.

COM1: Om en fil öppnas som "COM1:" så kan den skicka och ta emot data. Filer öppnade till "COM1:" kommunicerar med omvärlden genom datorns seriella port. Komplett syntax för "COM1:" instruktionen är:

```
COM1: [överföringshastighet] [,[paritet]
      [,[databitar] [,stoppbitar]]]]
```

överföringshastighet (baudrate) är den hastighet som Amigan sänder och tar emot data med. Följande hastigheter finns: 110, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600 och 19200.

paritet är ett sätt att kontrollera att dataöverföringen skett på korrekt sätt. Förvalt värde är "E". Man kan välja mellan "O" (som i engelskans Odd=udda), "E" (som i engelskans Even=jämn) eller "N" (som i engelskans None=ingen).

databitar är de bitar som innehåller riktiga data (till skillnad från paritets- och stoppbitar). Man har 5, 6, 7 eller 8 databitar.

stoppbitar används för att markera slutet på varje överfört ord. När överföringshastigheten är 110 så är förvalt värde för stoppbitar 2. För alla andra överföringshastigheter är förvalt värde 1. Om man valt två stoppbitar och fem databitar så används 1,5 stoppbitar.

Exempel:

```
OPEN "COM1:300,N,7,2" AS #1
```

Skrivare

Amiga Basic ger ett enkelt sätt att komma åt skrivare: Man adresserar skrivaren med "LPT1:".

Regler för filnamn

Det finns några regler man måste följa när man ger en fil ett namn. Alla filer har ett unikt namn med ett icke obligatoriskt skiv- eller skivenhetsnamn som prefix.

Filnamn

Amiga Basic:s sökvägsnamn kan vara 1-255 tecken långa och innehålla både versaler och gemena. Fil- och underbiblioteksnamnen i sökvägsnamnet kan vara upp till 30 tecken långa. Kontrolltecken får ej

användas i filnamn. Exempel på giltiga filnamn:

```
FAKTURA     A2400     MinFil
KONTROLLERA REGISTER
```

Skivspecifikation

Amigan har en inbyggd skivenhet men man kan också koppla till en extra skivenhet för att öka lagringskapaciteten. Man måste i sådana fall förklara för datorn vilken enhet som är aktiv, dvs vilken skivenhet datorn skall läsa och skriva filer på. Det gör man genom att lägga till filnamnet efter skivenhetens namn och åtskilja dem med ett kolon.

För att läsa in filer använder man "Open"-kommandot från Project-menyn. Om man vill läsa in en programfil som finns på en annan skiva än den i skivenheten så måste man naturligtvis först sätta in skivan som innehåller den önskade filen. Samma sak gäller om man vill spara en fil på en annan skiva än den i skivenheten.

Man kan också använda kommandona LOAD, MERGE, och RUN i resultatfönstret för att läsa in program från andra skivor än den i skivenheten. Man anger då kommandot, skivenhetens eller skivans namn och filnamnet. Man åtskiljer skivenhetens/skivans namn och filnamnet med ett kolon. Om man inte stoppat in skivan med programfilen i skivenheten så öppnas ett felfönster med meddelandet "Unknown Volume" (engelska: okänd skiva).

Handhavande av filer

Detta avsnitt ger grunderna i hur man sköter in- och utmatning i ett program.

Hantering av filer

I det följande ges en översikt av de kommandon och instruktioner som används för att hantera programfiler. Mer detaljerad information och syntaktiska regler ges i kapitel 8.

Öppna en programfil

Det finns huvudsakligen tre sätt att öppna en fil: med LOAD- och MERGE-kommandona samt CHAIN-instruktionen.

Vanligtvis använder man sig av LOAD. När man läser in en programfil så stängs alla öppna datafiler, minnet raderas från sitt innehåll och programmet som läses in placeras i minnet.

Ett annat sätt är att addera ett nytt program till det som finns i minnet. Man använder då MERGE-kommandot. Det är mycket användbart då man utvecklar stora program och vill testa enskilda delar av programmet separat. Man kan då testa och felsöka ett programs olika delar och sedan använda MERGE för att sätta ihop de enskilda delarna till ett större program.

En tredje väg är att starta ett program inifrån ett annat program. För att göra det används CHAIN-instruktionen. Med CHAIN så försvinner det första programmet från minnet och ersätts med det program som CHAIN kallade på. Man kan föra över variabelvärden från det första till det andra programmet.

Spara programfiler

Det finns huvudsakligen tre sätt att spara programfiler på: med "Save" och "Save As" från Project-menyn eller genom att skriva SAVE-kommandot i resultatfönstret.

För information om "Save" och "Save As" se "Menyraden och snabbval" i kapitel 3. För detaljer om SAVE, se kapitel 8.

Lagringsformatet är normalt binärt. Om man vill spara program i skyddad form så att de ej går att lista eller förändra, använd parametern P (står för engelska 'protected'=skyddad) tillsammans med SAVE-kommandot.

Om man vill spara en programfil i ASCII-format använder man parametern A (som i ASCII). ASCII-filer tar upp mer minnesutrymme på skiva men kan användas av ordbehandlare, vilket ofta kan vara praktiskt. CHAIN och MERGE kan bara arbeta med ASCII-filer.

Andra filkommandon

Med NAME-kommandot kan man döpa om program- och datafiler. Med KILL-kommandot kan man ta bort en fil från skivan. Se NAME och KILL i kapitel 8 för ytterligare information.

Datafiler - sekventiell och direkt I/O

Det finns två typer av datafiler som Amiga Basic kan skapa och läsa: sekventiella filer och direktåtkomstfiler (random access files).

Sekventiella filer

Sekventiella filer är enklare att skapa än direktåtkomstfiler men det går långsammare att ta fram data i dem. Data i en sekventiell fil är en serie av ASCII-tecken som ligger i en följd (sekventiellt) efter varandra i den ordning som de blivit inskrivna i filen. Data läses tillbaka från filen tecken för tecken i samma ordning som de skrivits in.

Varning: Sekventiella filer kan öppnas för läsning eller skrivning men inte för både läsning och skrivning samtidigt. Om man behöver lägga till data till en sekventiell fil som är stängd, öppna den inte för skrivning för då kommer den att raderas från sitt gamla innehåll innan ny data skrivs till den. Använd lägg-till-läget för att addera information till slutet av en existerande fil om du inte vill radera den redan existerande informationen i filen.

Man kan bestämma buffertstorleken på en sekventiell fil. Förvalt värde är 128 bytes (tecken). Man definierar storleken i OPEN-satsen när man öppnar filen. Den buffertstorlek man definierar är helt oberoende av längden på posterna som man läser eller skriver till filen. Stor buffert ökar I/O-hastigheten men stjäl utrymme från Amiga Basic. Liten buffert spar minne men sänker I/O-hastigheten.

Följande instruktioner används vid hantering av sekventiella filer:

CLOSE	LOF
EOF	OPEN
INPUT#	PRINT#
INPUT#	PRINT
WIDTH	LINE INPUT#
LOC	WRITE#

Skapa en sekventiell datafil

Program 1 är ett program som skapar en sekventiell fil, DATA, av information inmatad från tangentbordet.

Program 1 - skapar en sekventiell fil

```
OPEN "DATA" FOR OUTPUT AS #1
ENTER:
  INPUT "NAME ('DONE' TO QUIT)";N$
  IF N$="DONE" THEN GOTO FINISH
  INPUT "DEPARTMENT"; DEPT$
  INPUT "DATE HIRED";HIREDATE$
  WRITE #1,N$,DEPT$,HIREDATE$
  PRINT
GOTO ENTER
FINISH:
  CLOSE #1
END
```

Som syns i program 1 så krävs följande steg för att skapa en sekventiell fil och för att skriva data i den:

1. Öppna filen i skrivläge (så att filen är beredd att ta emot data.) med OPEN.

2. Skriv data i filen med WRITE#.

3. När all data är skrivna i filen, stäng filen med CLOSE.

Man kan justera datats format i filen med PRINT# USING- instruktionen. Satsen:

```
PRINT#1, USING"####.##,";A,B,C,D
```

kan användas för att skriva numeriska data i en fil med ett kommatecken mellan de olika talen. Kommat i slutet av strängen "####.##," gör att

värdena i filen kommer att separeras med ett kommatecken. Det är en god programmeringsvana att nyttja någon form av 'åtskiljare' mellan olika värden i en fil.

Om man vill ha kommatecken som åtskiljare mellan variabler kan man även använda WRITE#-instruktionen. Satsen

```
WRITE #1,A,B$
```

används för att skriva variablerna A:s och B$:s värden till en fil med ett kommatecken mellan de olika värdena.

Läsa data från en sekventiell fil

Program 2 läser data från filen "DATA" som skapades av program 1 och skriver ut namnen på de som anställdes under 1981.

Program 2 - läser en sekventiell fil

```
OPEN "I",#1,"DATA"
WHILE NOT EOF(1)
  INPUT #1,N$,DEPT$,HIREDATE$
  IF RIGHT$(HIREDATE$,2)="81" THEN PRINT N$
WEND
```

Program 2 läser varje post sekventiellt och skriver ut namnen på de som anställdes 1981. WHILE...WEND-slingan använder sig av EOF-funktionen för att kontrollera att alla data har lästs från filen. Programmet undviker därmed felet att försöka läsa förbi filslutet.

Lägga till data till en sekventiell fil

Om man har en sekventiell fil och vill lägga till mer information så kan man inte bara öppna filen för skrivning och sätta igång och skriva i den. Det är nämligen så att när man öppnar en sekventiell fil för skrivning så förstörs dess tidigare innehåll.

Man får istället använda sig av lägg-till-optionen ("A") i OPEN-satsen. Om filen inte existerar så kommer OPEN att fungera precis som om lägg-till inte hade specificerats.

Följande program lägger till data i en existerande fil med namnet "FOLKS".

Program 3 - lägg till data till en sekventiell fil

```
OPEN "A",#1,"FOLKS"
REM***Add new entries
NEWENTRY:
  INPUT "NAME";N$
  IF N$=""THEN GOTO FINISH 'Carriage Return
                                  exits loop
  LINE INPUT "ADDRESS ? ",ADDR$
  LINE INPUT "BIRTHDAY ? ",BIRTHDATE$
  PRINT #1, N$
  PRINT #1, ADDR$
  PRINT #1, BIRTHDATE$
  GOTO NEWENTRY
FINISH:
  CLOSE #1
END
```

LINE INPUT-instruktionen används för att hämta ADDR$ då den också tillåter inmatning av åtskiljningstecken (kommatecken och apostrofer.).

Direktåtkomstfiler

Det krävs fler programrader för att skapa och läsa en direktåtkomstfil än vad som krävs för en sekventiell fil. Det finns dock flera fördelar med en direktåtkomstfil framför en sekventiell fil. En fördel är att en direktåtkomstfil upptar mindre utrymme på skivan då AmigaBasic lagrar den i packat binärt format medan en sekventiell fil lagras som en serie ASCII-tecken. Den största fördelen med direktåtkomstfiler är, som namnet antyder, att data kan hämtas var som helst i filen. Man behöver inte, som med sekventiella filer, läsa igenom filen från början för att hitta en post. Informationen lagras nämligen i numrerade poster.

De instruktioner och funktioner som används vid hantering av direktåtkomstfiler är:

CLOSE	FIELD	MKD$	PUT
CVD	GET	MKI$	RSET
CVI	LOC	MKL$	
CVL	LOF	MKS$	
CVS	LSET	OPEN	

Skapa en direktåtkomstfil

Program 4 - skapar en direktåtkomstfil

```
OPEN "R",#1,"DATA",32
FIELD #1,20 AS N$,4 AS A$,8 AS P$
START:
  INPUT "2-DIGIT CODE (-1  TO QUIT)";CODE%
  IF CODE%=-1 THEN QUITFILE
  INPUT "NAME";PERSON$
  INPUT "AMOUNT";AMOUNT
  INPUT "PHONE";TELEPHONE$
  PRINT
  LSET N$=PERSON$
  LSET A$=MKS$(AMOUNT)
  LSET P$=TELEPHONE$
  PUT #1,CODE%
GOTO START
QUITFILE:
CLOSE #1
```

Som visas i program 4 så behöver man följande steg för att skapa en sekventiell fil:

1. Öppna OPEN filen som en direktåtkomstfil med OPEN. Frånvaron av läs-, skriv- och lägg-till-specifikationer i OPEN-satsen anger att det är en direktåtkomstfil. Om postlängden inte specificeras så antas postlängden vara 128 bytes (tecken).

2. Använd FIELD-instruktionen för att skapa utrymme i en direktåtkomstbuffert för de variabler som skall skrivas till direktåtkomstfilen. Bufferten är en reserverad plats i minnet som används för att överföra data mellan datafiler och programvariabler.

Exempel på hur man skapar en direktåtkomstbuffert med hjälp av FIELD:

```
FIELD #1,20 AS N$,4 AS ADDR$,8 AS P$
```

3. För att flytta data till bufferten används LSET-instruktionen. Numeriska värden måste göras om till strängar innan de kan placeras i bufferten. Man använder MKI$-funktionen för att konvertera heltal till strängar och MKS$-funktionen för konvertering av tal i enkel precision till strängar.

   Exempel på hur man flyttar data till bufferten:

```
LSET N$=X$
LSET ADDR$=MKS$(AMT)
LSET P$=TEL$
```

4. För att skriva data från bufferten till skivan, använd PUT-instruktionen och ange postnumret som ett uttryck.

Exempel:

```
PUT #1, CODE%
```

Program 4 läser information från tangentbordet och skriver det till en direktåtkomstfil. Varje gång PUT-satsen utförs så skrivs en post till filen. Det tvåsiffriga talet som matas in på rad 4 blir postnumret.

Läsning av data i direktåtkomstfiler

Program 5 läser data från direktåtkomstfilen DATA som program 4 skapade. Man matar in ett tvåsiffrigt tal och programmet letar rätt på den post som har detta postnummer och skriver ut posten på skärmen.

Program 5 - läs en direktåtkomstfil

```
OPEN "R",#1,"DATA",32
FIELD #1,20 AS N$,4 AS A$,8 AS P$
START:
  INPUT "2-DIGIT CODE (-1 TO QUIT)";CODE%
  IF CODE%=-1 THEN QUITFILE
  GET #1,CODE%
  PRINT N$
  PRINT USING "$$####.##";CVS(A$)
  PRINT P$: PRINT
  GOTO START
QUITFILE:
  CLOSE #1
```

Följande steg behövs för att läsa en direktåtkomstfil:

1. Öppna filen som direktåtkomstfil med hjälp av OPEN.

2. För att skapa utrymme i direktåtkomstbufferten för de variabler som skall läsas från filen, använd FIELD-instruktionen.

Anmärkning: I ett program som skriver och läser i samma direktåtkomstfil räcker det oftast med en OPEN-och en FIELD-instruktion.

3. För att läsa in den önskade posten till direktåtkomstbufferten, använder man GET-instruktionen.

Programmet kan nu använda datan som finns i bufferten. Numeriska värden som konverterats med MKI$- och MKS$-funktionerna måste konverteras tillbaka till tal. Man använder CVI-funktionen för heltal och CVS-funktionen för tal i enkel precision. MKI$- och CVI-funktionerna är varandras motsatser, samma sak gäller funktionerna MKS$ och CVS.

När LOC-funktionen används i samband med direktåtkomstfiler så returnerar den aktuellt postnummer dvs det postnummer som senast användes i en GET- eller PUT-sats. Satsen

```
IF LOC(1)>50 THEN END
```

avslutar ett program om den aktuella posten i fil nummer 1 har ett värde högre än 50.

Hantering av direktåtkomstfiler

Program 6 är ett inventeringsprogram som visar hur man kan hantera direktåtkomstfiler.

Program 6 - inventering

```
OPEN "INVEN.DAT" AS #1 LEN=39
FIELD #1,1 AS F$,30 AS D$,2 AS Q$ ,2 AS R$, 4 AS P$
FunctionLabel:
CLS:PRINT "Functions:":PRINT
PRINT "1. Initialize file"
PRINT "2. Create a new entry"
PRINT "3. Display inventory for one part"
PRINT "4. Add to stock"
PRINT "5. Subtract from stock"
PRINT "6. Display all items below reorder level"
PRINT "7. Done with this program"
PRINT:PRINT:INPUT "Function";FUNCT
IF (FUNCT>0) AND (FUNCT<8) THEN GOTO Start
GOTO FunctionLabel
Start:
ON FUNCT GOSUB 600,100,200,300,400,500,700
IF FUNCT<7 THEN GOTO FunctionLabel
END
100 :
   GOSUB PART
   IF ASC(F$)<>255 THEN INPUT  "Overwrite";confirm$
   IF ASC(F$)<>255 AND UCASE$(confirm$)<>"Y" THEN RETURN
```

```
   LSET F$=CHR$(0)
   INPUT "Description ";description$
   LSET D$=description$
   INPUT "Quantity in stock ";Quantity%
   LSET Q$=MKI$(Quantity%)
   INPUT "Reorder level ";reorder%
   LSET R$=MKI$(reorder%)
   INPUT "Unit price ";price
   LSET P$=MKS$(price)
   PUT #1,PART%
   INPUT "Press RETURN to continue",DUM$
   RETURN
200 :
   GOSUB PART
   IF ASC(F$)=255 THEN GOSUB NullEntry:RETURN
   PRINT USING "Part Number ###";PART%
   PRINT D$
   PRINT USING "Quantity on hand #####";CVI(Q$)
   PRINT USING "Reorder level #####";CVI(R$)
   PRINT USING "Unit price $$##.##";CVS(P$)
   INPUT "Press RETURN to continue",DUM$
   RETURN
300 :
   GOSUB PART
   IF ASC(F$)=255 THEN GOSUB NullEntry:RETURN
   PRINT D$
   PRINT "Current quantity: ";CVI(Q$)
   INPUT "Quantity to add";additional%
   Q%=CVI(Q$)+additional%
   LSET Q$=MKI$(Q%)
   PUT #1,PART%
   RETURN
400 :
   GOSUB PART
   IF ASC(F$)=255 THEN GOSUB NullEntry:RETURN
   PRINT D$
425 :
   INPUT "Quantity to subtract";less%
   Q%=CVI(Q$)
   IF (Q%-less%)<0 THEN PRINT "Only ";Q%;" in stock":GOTO 425
   Q%=Q%-less%
   IF Q%<=CVI(R$) THEN PRINT "Quantity now ";Q%
   LSET Q$=MKI$(Q%)
   PUT #1,PART%
   INPUT "Press RETURN to continue",DUM$
   RETURN
500 :
   reorder=0
   FOR I=1 TO 100
   GET #1,I
   IF ASC(F$)=255 GOTO 525
   IF CVI(Q$)<CVI(R$) THEN PRINT D$;" Quantity";CVI(Q$);TAB(30)
   IF CVI(Q$)<CVI(R$) THEN PRINT "Reorder level ";CVI(R$)
   IF CVI(Q$)<CVI(R$) THEN reorder=(-1)
```

```
525 :
   NEXT I
   IF reorder=0 THEN PRINT "All items well-stocked."
   INPUT "Press RETURN to continue",DUM$
   RETURN
600 :
   INPUT "Are you sure";confirm$
   IF confirm$<>"y" AND confirm$<>"Y" THEN RETURN
   LSET F$=CHR$(255)
   FOR I=1 TO 100
   PUT #1,I
   NEXT I
   RETURN
PART:
Enterno:
   INPUT "Part number? ",PART%
   IF (PART%<1) OR (PART%>100) THEN PRINT "bad part number"
   IF (PART%<1) OR (PART%>100) THEN GOTO Enterno
   GET #1,PART%
   RETURN
NullEntry:
    PRINT "Null Entry."
    INPUT "Please press RETURN",DUM$
    RETURN
700 : CLOSE #1
    RETURN
```

Överföra data mellan Amiga Basic och ordbehandlingsprogram

De flesta ordbehandlingsprogram använder en mängd kontrolltecken i sina filer för styrning av format, skrivare etc. Dessa kontrolltecken syns inte på skärmen. Kontrolltecknen kan förstöra en fil som är avsedd att användas som programfil, t ex ett Amiga Basic-program. I de flesta ordbehandlingsprogram finns ett filformat kallat 'Text bara', 'Oformaterad text', 'Icke-dokument' eller något liknande. När man sparar en textfil med ett sådant format så tas alla kontrolltecken bort. Bara själva texten sparas. Spara dina programfiler i detta format så kan Amiga Basic tolka dem.

Om man vill bearbeta sitt program i en ordbehandlare så måste man spara det med SAVE,A i resultatfönstret. Programmet sparas då i ASCII-format vilket kan läsas av de flesta ordbehandlingsprogram.

6 Avancerad programmering

I detta kapitel skall vi gå igenom en del finesser i Amiga Basic som t ex underprogram, minneshantering och avbrottshantering.

Underprogram

Underprogram liknar i mångt och mycket subrutiner men det finns tre stora fördelar med underprogram framför subrutiner:

1. Variablerna i underprogram är isolerade från variablerna i resten av programmet. Det betyder att om en variabel i ett underprogram har samma namn som en variabel i huvudprogrammet så kommer de ändå att returnera skilda värden. Man kallar variablerna i underprogrammet för lokala, vilket betyder att deras värde inte kan påverkas utanför underprogrammet.

2. Som programmerare upptäcker man snart att man ofta sitter och skriver om samma rutin om och om igen till olika program. Istället för att lägga ner arbete på något som man redan gjort en gång så skriver man helt enkelt dessa rutiner som underprogram. Man kan skapa sig hela bibliotek med underprogram som med små förändringar enkelt adderas till huvudprogrammet med MERGE-kommandot. På så sätt kan mycket programmeringstid sparas.

3. Ett underprogram kan inte startas av misstag p g a något trivialt programmeringsfel. En subrutin kan bli utförd om det inte finns någon GOTO-sats ovanför den. Underprogram utförs om, och endast om, det finns ett anrop till dem.

Anrop till underprogram

Man anropar en underprogram med instruktionen CALL följt av en lista med parametrar (se CALL i kapitel 8 för ytterligare information). Nedan kommer vi att använda uttrycken formella parametrar och aktuella parametrar. Aktuella parametrar refererar till de variabler som ingår i CALL-instruktionen.

Exempel:

```
CALL KALKSKATT(DELTOTAL, SKATT, TOTAL())
```

I exemplet så är DELTOTAL, SKATT och den indexerade variabeln TOTAL aktuella parametrar.

Formella parametrar refererar till de parallella värden som underprogrammet använder. Underprogrammet KALKSKATTS första programrad skulle kunna se ut så här:

```
SUB KALKSKATT(KALKYL, SKATT2, SUMMA(1)) STATIC
```

De formella parametrarna är här variablerna KALKYL, SKATT2 och den indexerade variabeln SUMMA. De formella parametrarna returnerar värden till huvudprogrammets aktuella parametrar DELTOTAL, SKATT och TOTAL().

En aktuell parameter får sitt värde genom att den är "knuten" till en formell parameter. När den formella parametern ändrar värde i underprogrammet så kommer motsvarande aktuella parameter också att ändra värde.

Exempel:

```
CALL SUMMERA(A,B,C)
          .
          .
          .
          .
SUB SUMMERA(X,Y,Z) STATIC
   Z=X+Y
   X=X+12
   Y=Y+94
END SUB
```

Om A=2 och B=3 när CALL utförs så blir A=14 och B=97 när huvudprogrammet återfår kontrollen. Aktuella parametern A är knuten till den formella parametern X och B är knuten till Y. Om X:s värde förändras i underprogrammet så förändras A:s värde i huvudprogrammet också.

Om man inte vill att ett underprogram skall ändra värdena på variablerna i huvudprogrammet men ändå vill använda dem som indata till underprogrammet så går det bra. Man sätter då helt enkelt parenteser kring de variabler i CALL-satsen som skall förbli opåverkade oavsett vad som händer i underprogrammet.

Exempel:

```
CALL SUMMERA((A), (B), Resultat)
```

Parenteserna kring A och B gör att Amiga Basic betraktar dem som uttryck och därför kan deras värden inte förändras av underprogrammet. Man behöver dock inte använda parenteser kring riktiga uttryck som skall användas som indata till underprogram.

Exempel:

```
CALL SUMMERA(1+2,3*A,Resultat)
```

Observera att de aktuella och formella parametrarna måste vara av samma typ för att fel inte skall uppstå.

Exempelvis skulle

```
CALL KALKYL(1)
      .
      .
      .
SUB KALKYL (X) STATIC
```

inte fungera eftersom man försöker sätta heltalet 1 till X som är en variabel med enkel precision.

Rätt är däremot:

```
CALL KALKYL(1.0)
      .
      .
      .
SUB KALKYL(X) STATIC
```

Underprogrammets avgränsare

Underprogram inleds alltid med SUB-instruktionen och avslutas alltid med END SUB. Man använder sig av EXIT SUB för att hoppa ur ett underprogram innan man nått END SUB. När END SUB eller EXIT SUB utförs så hoppar programmet tillbaka till den rutin som utförde CALL-instruktionen.

Den kompletta syntaxen för ett underprogram är:

```
SUB underprogramnamn [(lista på formella parametrar)] STATIC
   [SHARED lista på variabler]
         .
         .
         .
         .
END SUB
```

Underprogramnamnen kan vara upp till 40 tecken långa och bestå av alfanumeriska tecken men måste ha en bokstav som första tecken. Man får inte använda samma namn på flera underprogram.

Listan på formella parametrar kan bestå av enkla eller indexerade variabler. Parametrar separeras från varandra med kommatecken. Antalet parametrar begränsas bara av radens maximala längd.

STATIC-instruktionen anger att variablerna i underprogrammet endast kan få och förändra värde i underprogrammet. STATIC-variablers värde kan inte påverkas utanför underprogrammet, de är lokala variabler. Då man använder STATIC-instruktionen får inte underprogrammet vara rekursivt, dvs det får inte innehålla anrop till sig självt eller anrop till underprogram som i sin tur innehåller anrop till det ursprungliga underprogrammet.

Variabler som deklareras som SHARED kan påverkas utanför underprogrammet; de är delade. Alla variabler är lokala tills de definieras som SHARED. Se SHARED i kapitel 8 för ytterligare information.

Man kan använda alla utom följande Amiga Basic-kommandon i underprogram:

* Definition av funktioner (användardefinierade funktioner).

* SUB...SUB END, dvs man får inte definiera ett underprogram inuti ett annat underprogram. Definitioner av underprogram kan inte nästlas.

* COMMON-instruktioner.

* CLEAR-instruktioner.

Deklarera en indexerad parameter

Enkla variabelparametrar kan ha vilket giltigt namn som helst. En indexerad parameter måste deklareras så här:

variabelnamn ([*antal element*])

antal element anges med ett heltal. *variabelnamn* byggs upp på samma sätt som för enkla variabler.

Exempel:

```
SUB MATADD2(N%,M%,A(2),B(2),C(3)) STATIC
      .
      .
      .
END SUB
```

I exemplet ovan så är N% och M% heltalsvariabler, A() och B() är indexerade variabler med två element vardera och C() är en indexerad variabel med tre element.

Enkla variabler och element i indexerade variabler

Följande program visar hur man överför värden från ett underprogram till ett huvudprogram. Överföringen sker på samma sätt oavsett om parametrarna är enkla variabler, element ur en indexerad variabel eller hela indexerade variabler.

Exempel:

```
A=5:B=2
CALL KVADRAT(A,B)
PRINT A,B
END
SUB KVADRAT(X,Y) STATIC
   Y=X*X
END SUB
```

Exemplet kommer att skriva ut resultaten 5 och 25 på skärmen. Varje referens till Y blir en referens till B och varje referens till X blir en referens till A. Med andra ord, varje gång KVADRAT använder sig av Y så använder den sig egentligen av B.

Uttryck som aktuella parametrar

Man kan också använda uttryck som indata till underprogram. Man kan använda alla uttryck som är giltiga i Amiga Basic som aktuella parametrar till ett underprogram. När Amiga Basic stöter på ett uttryck i en CALL-sats i en parameterlista så sätts dess värde till en temporär variabel som sedan överförs till underprogrammet med variabelreferens. Det fungerar på samma sätt som överföring av värden till funktioner.

Om en enkel eller indexerad variabel omges av parenteser i CALL-satsen så överförs de på samma sätt som uttryck, dvs som anrop-med-värde. Om vi exempelvis ändrar CALL KVADRAT i ovanstående exempel till

```
CALL KVADRAT(A,(B))
```

så skulle resultatet som skrivs ut på skärmen bli 5 och 2. B överförs nu som ett värde och kan inte påverkas av underprogrammet.

Anmärkning: Indexerade variabler skall inte överföras som parametrar på det sätt som som beskrivits ovan när man anropar maskinspråksrutiner. Istället så överförs baselementets adress i minnet till rutinen med hjälp av en variabelreferens. Maskinspråksrutinen kan då komma åt alla element i den indexerade variabeln.

Exempel:

```
CALL X(VARPTR(A(0,0)))
```

Delade och lokala variabler i underprogram

Man kan definiera variabler till att vara lokala inom en rutin eller till att vara delade.

Delade variabler

Delade variablers värden kan man komma åt i ett underprogram utan att behöva skicka med värdet som en parameter vid anropet.

Med SHARED-instruktionen i ett underprogram kan man definiera huvudprogrammets variabler till att bli delade så att man kan använda

dem i underprogrammet. SHARED-instruktionen påverkar bara variabler i sitt eget underprogram. Bara för att man deklarerat en variabel till att vara delad i underprogrammet A så blir den inte automatiskt delad i underprogrammet B.

Exempel:

```
LET A=1:LET B=5:LET C=10
DIM P(100),Q(100)
      .
      .
      .
      .
SUB AMIGA STATIC
   SHARED A,B,P(),Q()
   .
   .
   .
   .
END SUB
```

I exemplet så är alla variabler utom C delade.

Lokala variabler

Som vi tidigare nämnt så är alla variabler lokala som inte deklarerats till att vara delade. De kan inte ändras av satser utanför underprogrammet. De initieras till det numeriska värdet noll (0) eller till en tom sträng. Om underprogrammet har körts och avslutats någon gång så behåller alla variabler de värden som de hade när underprogrammet avslutades.

Indexets övre och undre gräns hos indexerade variabler

Man kan ta reda på indexets gränser hos en indexerad variabel med instruktionerna LBOUND (undre gräns) och UBOUND (övre gräns).

LBOUND returnerar indexets undre gräns vilket är 0 eller 1 beroende på hur OPTION BASE-instruktionen är använd. Förvalt värde är 0. UBOUND ger indexets övre gräns. Funktionerna har två syntaxer: En generell syntax och en enklare syntax som man använder för endimensionella variabler:

LBOUND (*variabelnamn*) - för endimensionella variabler
LBOUND (*variabelnamn, dimension*) - för flerdimensionella variabler

UBOUND (*variabelnamn*) - för endimensionella variabler
UBOUND (*variabelnamn, dimension*) - för flerdimensionella variabler

variabelnamn är den indexerade variabelns namn och *dimension* är ett heltal mellan 1 och antalet definierade dimensioner hos variablerna.

Exempel:

```
DIM P(3,1,5)

PRINT UBOUND(P,1)
PRINT UBOUND(P,2)
PRINT LBOUND(P,3)
```

Programsnutten ovan skriver ut 3, 1 och 1 på skärmen.

LBOUND och UBOUND är speciellt användbara för att ta reda på storleken hos en indexerad variabel som man skickat till ett underprogram. Se LBOUND i kapitel 8 för utförligare förklaringar och exempel.

Avbrottshantering

I Amiga Basic kan man ta hand om avbrott i realtid, dvs oavsett vilken sats programmet befinner sig på så kan det reagera på en speciell händelse. De avbrott (händelser) Amiga Basic kan reagera på är:

* Tid (ON TIMER).

* Försök att avbryta programmet (ON BREAK).

* Val av en meny (ON MENU) som man själv skapat.

* Klickning av musen (ON MOUSE).

* Objektkollision (ON COLLISION).

Man utnyttjar avbrottshanteringen på så sätt att man skriver en subrutin som utförs då avbrottet uppkommer. Subrutinen tar hand om avbrottet.

När subrutinen är färdig och utför sin RETURN-instruktion så återvänder programmet till programraden efter den programrad som utfördes före

Det här avsnittet ger en översikt över hur avbrottshantering går till. För mer detaljer se kapitel 8.

Avbrottshantering styrs av följande instruktioner:

avbrottstyp ON	sätter på avbrottshantering för angiven avbrottstyp.
avbrottstyp OFF	stänger av avbrottshantering för angiven avbrottstyp.
avbrottstyp STOP	ger paus i avbrottshantering för angiven avbrottstyp.

avbrottstypen kan vara någon av följande:

TIMER	ger avbrott då ett visst antal sekunder förflutit.
MOUSE	ger avbrott då musen klickas.
MENU	ger avbrott då en meny väljs.
BREAK	ger avbrott då användaren försöker avbryta programmet med Amiga-punkt, Esc eller Ctrl-C. Om BREAK är "ON" så avslutas inte programmet förrän en END-instruktion utförs. Detta kan leda till att programmet hamnar i en evig slinga som inte kan avbrytas med mindre än att man stänger de aktuella resultat- och List-fönstren.
COLLISION	ger avbrott då objekt skapade av OBJECT.SHAPE-instruktionen kolliderar med något annat objekt eller med en fönsterlist.

ON...GOSUB

ON...GOSUB-instruktionen ger hopp till en subrutin som behandlar ett avbrott. Syntaxen är:

ON *avbrottstyp* GOSUB radnummer

Om *radnummer* är noll (0) så kopplar man bort avbrottshanteringen för den angivna avbrottstypen.

Aktivera avbrottshantering

Avbrottshantering sätts på genom att de båda satserna

avbrottstyp ON
ON *avbrottstyp* GOSUB *radnummer*

utförs. Avbrottstypen för de båda satserna måste vara densamma och *radnummer* måste vara större än noll (0).

Avstängning av avbrottshantering

Med satsen

avbrottstyp OFF

stängs avbrottshanteringen av för en viss avbrottstyp.

Med

ON *avbrottstyp* GOSUB 0

kan man också avsluta avbrottshanteringen.

Paus i avbrottshantering

När avbrottshanteringen avbryts med

avbrottstyp STOP

så kommer ingen avbrottshantering att ske. Förekomsten av avbrottstypen lagras så att så fort som avbrottshanteringen sätts på igen så kommer, om avbrott förekommit under stopptiden, subrutinen som hanterar avbrottet att utföras.

När avbrottshantering utförs så utförs automatiskt ett STOP för den avbrottstypen så att rekursiv avbrottshantering inte uppstår. RETURN-instruktionen i avbrottssubrutinen sätter på avbrottshanteringen igen.

Anmärkning: då ett avbrott orsakats av att ett fel hanteras så stängs all avbrottshantering av den typen av tills en RESUME-instruktion utförs.

Minneshantering

I Amiga Basic kan man fördela minnet för olika ändamål med hjälp av CLEAR-instruktionen. Det är till stor hjälp när man skriver stora program.

Man kan kontrollera tre olika minnesområden med CLEAR:

* Stackminnet.
* Amiga Basic-minnet.
* Systemminnet.

Stackminnet

I stacken lagras adresser dit programmet skall återvända efter GOSUB-anrop, nästlade underprogramanrop, nästlade FOR...NEXT slingor, nästlade WHILE...WEND-slingor och programdefinierade funktioner.

Spara stackminne

En del Amiga ROM-anrop tar en ansenlig mängd av stackminnet i anspråk. Ju mer nästlade anrop som man använder, desto mer stackminne krävs.

Amiga Basic-minnet

Amiga Basic-minnet innehåller programtexten, numeriska variabler, strängvariabler och bufferten för öppnade filer.

Spara Amiga Basic-minne

En sekventiell fil har en förvald buffert på 128 bytes (tecken). Man kan spara minne genom att minska denna buffertstorlek men det kan minska hastigheten hos program som använder mycket in-och uthantering. Se OPEN i kapitel 8 för mer detaljer om hur man definierar storleken på bufferten till sekventiella filer.

Datatypen på de numeriska variabler som används i ett program påverkar också minnesåtgången. Heltal tar upp hälften så mycket minne som tal i enkel precision, vilka i sin tur tar upp hälften så mycket minne som tal i dubbel precision.

Med CHAIN-instruktionen kan man spara minne. Genom att dela upp program i flera små program som läses in vid behov av CHAIN behöver man inte ha hela programmet i minnet.

Systemminnet

Systemminnet innehåller bufferten för SOUND- och WAVE-information vilka behöver, när de används, 1024 bytes (tecken) RAM. LIBRARY-, WINDOW- och SCREEN-instruktionerna använder också minne från systemminnet. Amiga Basic delar systemminnet med andra rutiner som körs på Amigan.

Spara systemminne

Man kan spara systemminne genom att ta bort SOUND/WAVE-bufferten när den inte längre behövs. Använd instruktionen WAVE 0.

Använd CLEAR för att fördela minne

Man aväander CLEAR-instruktionen för att fördela minne mellan de tre minnesområdena. CLEAR har följande syntax:

CLEAR [, [*Amiga Basic minnets storlek*][*stackminnets storlek*]]

Amiga Basic-minnet är ett numeriskt värde som anger hur många bytes (tecken) som Amiga Basic-minnet skall uppta; *stackminnet* anger hur många bytes som skall reserveras för stacken.

Kvarvarande minne (totalt minne minus Amiga Basic-minnet minus stackminnet) är det minnesutrymme som blir reserverat för systemminnet.

FRE-funktionens användning

Använd FRE-funktionen för att ta reda på kvarvarande minne i de olika minnesområdena. Syntaxen för FRE är:

FRE*(n)*

eller

FRE*(" ")*

n kan ha tre värden:

1. n=-1 gör att FRE returnerar antalet fria bytes i systemminnet.

2. n=-2 gör att FRE returnerar antalet bytes som **aldrig** använts av stackminnet. FRE(-2) ger sålunda inte antalet fria bytes i stacken. Man använder FRE(-2) för att testa minneskonsumtionen hos olika program för att sedan kunna justera stackstorleken med CLEAR.

3. Om n är skilt från -1 och -2 eller om man använder FRE(" ")-funktionen så får man antalet fria bytes i Amiga Basic-minnet i retur.

Alla versioner av FRE komprimerar blanktecken i strängar.

7 Skapa animerade bilder

I detta kapitel skall vi gå igenom objekteditorn som är ett hjälpprogram för att skapa bilder som senare kan manipuleras med Amiga Basic-instruktioner.

Översikt av objekteditorns möjligheter

Amiga Basic använder två sätt för att komma åt Amigans bildmöjligheter: objekteditorn och Amiga Basic:s instruktioner för styrning av bilder.

Instruktionerna COLLISION och OBJECT (beskrivs utförligare i kapitel 8) används för att manipulera bilder i resultatfönstret. Objekteditorn används för att skapa dessa bilder (eller *objekt* som vi hädanefter kommer att kalla dem).

Med objekteditorn kan man:

* Snabbt skapa ellipser, rektanglar och linjer med hjälp av musen. Figurerna ritas upp på objekteditorns *duk* som är den del av objekteditorn som man skapar objekt på.

* Rita på fri hand på duken med objekteditorns penna.

* Välja färg på ramen som omger ett objekt.

* Måla objektens insida med samma färg som ramen.

* Ändra och ta bort objekt.

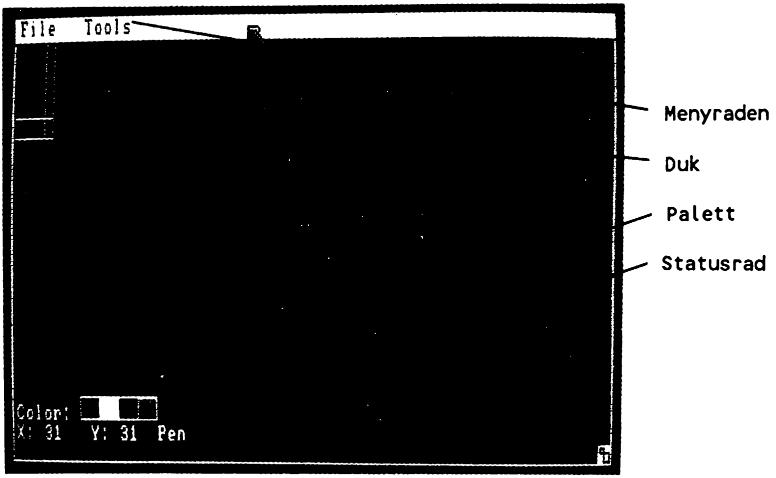
Man sparar ett objekt i en fil som kommer att innehålla definitionen av objektets utseende (t ex storlek, utseende, färg). För att använda objektet i ett program skall man öppna filen, läsa in dess innehåll som en sträng och använda OBJECT.SHAPE-instruktionen. I kapitel 8 finns ett exempel på hur det går till att använda OBJECT.SHAPE-instruktionen.

Anmärkning: Objekteditorn tilldelar automatiskt de objekt man skapar vissa egenskaper vilket gör att vid programkörning så kommer objekten att både kollidera med sig själva och med fönstrets lister. Man kan ändra

denna inställning med en OBJECT.HIT-instruktion, se kapitel 8 för mer detaljer.

Objekteditorns fönster

Objekteditorns fönster ser ut så här:



Menyraden

Menyraden innehåller tre menyer:

* File-menyn som man använder för att spara objekt och för att hämta redan existerande objekt för redigering.

* Tools-menyn som innehåller de verktyg som behövs när man skapar objekt.

* Enlarge-menyn som används för att tillfälligt förstora objektet så att man kan rita fina detaljer.

Duken

Duken finns uppe i vänstra hörnet av skärmen. Det är där man skapar och tar bort objekt.

Man kan öka storleken på duken genom att peka på storleksrutan, trycka ner och hålla nere musens valknapp, flytta musen tills duken får önskad storlek och sedan släppa upp valknappen.

När man skapar sprites (en av de två typer av objekt som man kan skapa, beskrivs utförligare nedan) så kan duken inte göras bredare än den är när man startar programmet (16 bildelement, från 0 till 15). Duken kan dock göras högre.

Paletten

Paletten används för att välja den färg man skall rita med och för att välja färgen på objektets ram. För att välja färg, klicka den ruta på paletten som har den önskade färgen. Tecknen i ordet *Color* byter då färg för att indikera vilken färg man har valt.

Antalet olika färger som man kan välja på paletten beror på vilket skärmdjup man har, vilket definieras med SCREEN-instruktionen, se kapitel 8.

För att skapa objekt med mer än fyra färger så måste man ändra i objekteditorns programlistning. I programlistningen finns kommentarer inskrivna som beskriver hur man gör detta.

Det programmet som skall använda sig av objektet måste använda samma skärmdjup som det definierade objektet använder.

Statusraden

På statusraden anges:

* Vilken typ (sprites eller bobs) av objekt som redigeras.

* Dukens storlek uttryckt i x och y koordinater.

* Det verktyg som används.

Verktygen är:

Pen	penna
Oval	ritar ellipser eller cirklar
Line	ritar linjer
Rectangle	ritar rektanglar
Paint	målar insidan av slutna figurer
Erase	radergummi

Editorns menyer

I de följande avsnitten ger vi en sammanfattning av innehållet hos de tre menyer som objekteditorn har.

File-menyn

Kommando	**Funktion**
New	Rensar skärmen och återställer duken till dess förvalda storlek.
Open	Öppnar och läser in en redan existerande fil innehållande definitionen för ett objekt. Ange namnet på filen som skall läsas in och tryck <RETUR>.
Save	Sparar en fil under samma namn som man angav när man öppnade den. Om man skapat en ny fil så ber "Save" att man skall ange det namn som filen skall sparas under. Ange namnet och tryck <RETUR>.
Save as	Samma som "Save" men man tillåts spara en redan existerande fil under nytt namn. Ange nytt namn och tryck <RETUR>.
Quit	Avslutar objekteditorn och återvänder till Amiga Basic:s resultatfönster.

Tools-menyn

Kommando	Funktion
Pen	Används när man vill rita på fri hand.
Line	Drar en rak linje mellan två punkter.
Oval	Ritar ellipser eller cirklar. Man bestämmer själv höjd och bredd.
Rectangle	Ritar en rektangel. Man bestämmer själv höjd och bredd.
Erase	Radergummi.
Paint	Målar insidan av slutna figurer med den aktuella färgen.

Enlarge-menyn

Kommando	Funktion
4x4	Ökar dukens storlek med faktor fyra. Duken får inte vara bredare än 100 bildelement eller högre än 31 bildelement för att funktionen skall kunna fungera.
1x1	Återställer dukens storlek till normal storlek.

Bobs och sprites

Det finns två typer av objekt hos Amigan: *bobs* och *sprites*. När man startar objekteditorn så frågar den om man vill definiera en bob eller en sprite.

Det finns ett antal viktiga skillnader mellan bobs och sprites. Här nedan listar vi dem:

Bobs	Sprites
Rör sig långsammare än sprites.	Rör sig snabbare än bobs.

Storleken är endast begränsad av minnets storlek.	Bredden får högst vara 16 bildelement.
Alla färger får användas.	Endast tre färger får användas.
Alla bobs kan visas på samma gång.	Endast fyra sprites kan visas samtidigt på samma rad.
Alla typer av skärmdjup kan användas.	Skärmdjupet måste vara två (2).

Skärmdjupet sätts med SCREEN-instruktionen, se kapitel 8.

För mer detaljerad beskrivning av bobs och sprites, se "Graphics Animation" i *Amiga ROM Kernel Manual.*

Hur man skapar objekt

Objekteditorn finns på Extras-skivan i BasicDemos-lådan under namnet ObjEdit. Man startar editorn precis på samma sätt som man startar andra Amiga Basic-program (kapitel 2 beskriver det). Om man vill öppna objekteditorn inifrån Amiga Basic används filnamnet BasicDemos/ObjEdit.

Anmärkning: om man använder en Amiga med 256 kb minne så bör man flytta objekteditorns ikon ut ur BasicDemos-fönstret till Workbench-fönstret. Alla fönster stängs sedan och objekteditorns ikon dubbelklickas. Detta gör att man får största möjliga minne fritt för objekteditorns användning.

Här nedan kommer vi ge instruktioner steg för steg hur man skapar ett objekt.

Stegvis beskrivning av hur man skapar objekt.

1. När man startat objekteditorn kommer följande upp:

```
Enter 1 if you want to edit sprites
```

```
Enter 0 if you want to edit bobs >
```

Tryck 1 och <RETUR> om du vill ändra/skapa en sprite. Tryck 0 om du vill ändra/skapa en bob.

Anmärkning: använd inte fönsterväxlare bak för att flytta objekteditorns fönster bakom de övriga fönsterna.

2. Objekteditorns fönster kommer upp. Välj "New" från File-menyn om du vill skapa ett nytt objekt. Välj "Open" om du vill modifiera ett redan existerande objekt.

3. Välj det verktyg du vill använda från Tools-menyn.

   För att rita eller radera, flytta pekaren till startpositionen på duken. Tryck ner och håll nere musens valknapp, flytta pekaren till slutpositionen och släpp upp valknappen.

   Observera att när man skapar en ellips eller en cirkel så kommer först en rektangel upp på duken, men så fort man släpper upp valknappen så ersätts den av en cirkel eller av en ellips.

4. För att byta färg, peka på önskad färg på paletten och klicka valknappen.

5. För att måla insidan av en figur, välj först önskad färg från paletten, välj "Paint" från Tools-menyn, peka på det område du vill måla och klicka slutligen valknappen.

   Området som du vill måla med en färg måste vara helt avgränsat av en linje med samma färg. Annars (om det finns hål i linjen) så kommer färgen att läcka ut till omgivningen.

6. För att göra duken större, peka på storleksrutan, tryck och håll ned valknappen och flytta musen. När duken har önskad storlek, släpp upp valknappen.

   Amiga Basic betraktar en duk som ett objekt, oavsett hur många olika figurer man målat på den. Vill man ha flera objekt i ett program så måste de skapas på separata dukar och sparas i separata filer.

7. När du är färdig med ett objekt, välj "Save As" från File-menyn om du har skapat ett nytt objekt och "Save" om du har redigerat ett gammalt objekt.

8 Amiga Basic: referensdel

Första delen av detta kapitel beskriver:

* Amiga Basic:s byggstenar, dvs de tecken man kan använda.
* Amiga Basic:s syntax och grammatik.

Andra delen är en komplett beskrivning av alla instruktioner och funktioner. De beskrivs i bokstavsordning.

Teckenuppsättningen

Amiga Basic:s teckenuppsättning består av bokstäver, siffror och specialtecken. De är de enda tecken som Amiga Basic-tolken känner igen och påverkas av. Det finns många fler tecken som man kan visa på skärmen eller skriva ut på en skrivare men de påverkar inte tolkens funktion.

De **bokstäver** som Amiga Basic känner igen är alla versaler och gemener i det amerikanska engelska alfabetet (A-Z, a-z).

Siffrorna är 0 - 9.

Följande **specialtecken** ingår i Amiga Basic:s teckenuppsättning:

Tecken	**Namn eller funktion**
	Blank eller mellanslag
=	Lika med eller tilldelningstecken
+	Plustecken
-	Minustecken
*	Asterisk eller multiplikationstecken
/	Snedstreck eller divisionstecken
^	Pil upp eller exponentieringstecken
(	Vänsterparentes
)	Högerparentes

%	Procenttecken
#	Brädgård eller nummertecken
$	Dollartecken
!	Utropstecken
[	Vänster hakparentes
]	Höger hakparentes
,	Komma
.	Punkt (även decimalpunkt)
'	Apostrof
;	Semikolon
:	Kolon
&	"Och"-tecken
?	Frågetecken
<	Mindre än
>	Större än
\	Omvänt snedstreck eller heltalsdivisionstecken
@	Kommersiellt A
_	Understrykningstecken
<RETUR>	Avslutar inmatning av teckenrad
"	Citationstecken

Följande **tangentkombinationer** används i Amiga Basic:

Tangentkombination	**Funktion**
Amiga-punkt	Avbryter programkörning och återvänder till kommandonivån
Amiga-S	Stannar (Suspenderar) programkörning
Amiga-T	Kör nästa sats i ett program
Amiga-C	"Copy"-kommandot från Edit-menyn
Amiga-P	"Paste"-kommandot från Edit-menyn
Amiga-X	"Cut"-kommandot från Edit-menyn
Amiga-R	"Start"-kommandot från Run-menyn
Amiga-L	"Show List"-kommandot från Windows-menyn

Programrader

En programrad i Amiga Basic har följande syntax:

[nnnnn] sats [:sats...][kommentar]<RETUR>

eller

[alfanumerisk adress:] sats [:sats...][kommentar]<RETUR>

nnnnn (radnumret) är ett heltal mellan 0 och 65529

Den *alfanumeriska adressen* är en kombination av bokstäver, siffror och punkter som inleds med en bokstav och avslutas med ett kolon (:).

kommentar är en ej körbar sats som man kan använda för att lägga in kommentarer och upplysningar om programmets funktion vilket gör programkoden lättare att förstå.

Sammanfattningsvis så kan en programrad börja med ett radnummer, en alfanumerisk adress, inget av dem eller både radnummer och adress. Programraden slutar med <RETUR> vilket är ett osynligt tecken som markerar slutet på raden. Programraden kan maximalt bestå av 255 tecken. Man kan placera flera satser på samma rad om man åtskiljer dem med kolon.

Radnummer och adresser används för att göra program mer lättlästa eller för att ändra programflödet exempelvis vid GOSUB-instruktioner. Om man t ex vill att en rutin bara skall utföras när ett speciellt villkor är sant så skulle man kunna skriva:

```
IF Namn$<>"" THEN GOSUB Namnlas
```

Programmet hoppar till den subrutin som börjar med adressen Namnlas. Observera att inget kolon behövs efter Namnlas i GOSUB-satsen.

Anmärkning: Amiga Basic-tolken utför programraderna i den ordning de blivit inmatade i editorn, dvs inte efter storleken på radnumren. Andra Basic-tolkar sorterar ofta programraderna efter stigande nummer, men så är inte fallet i Amiga Basic.

Definition av adresser

Alfanumeriska adresser kan innehålla 1 - 40 tecken som kan vara bokstäver, siffror och punkter. Adressen måste börja med en bokstav. Adresser kan med fördel användas för att göra program lättare att läsa.

Exempel på giltiga radnummer och adresser:

Radnummer	Adress
100	ALPHA:
6500	A16:
	SCREEN.SUB

Restriktioner

För att tolken skall kunna skilja mellan adresser och variabler så måste varje adress avslutas med ett kolon. Man får inte ha ett mellanslag mellan adress och kolon. När man refererar till en adress i en GOSUB eller GOTO-sats eller i andra styrsatser så skall inte kolonet tas med. Man kan inte använda reserverade ord som adresser.

Man kan använda radnummer 0 men avbrottshanterande satser använder 0 för att stänga av avbrottshantering. I satsen

```
ON ERROR GOTO 0
```

så kommer inte ett hopp till rad 0 ske om ett fel uppstår. Istället så kommer avbrottshanteringen att stängas av när satsen utförs.

Observera att Amiga Basic egentligen betraktar radnummer som en typ av adresser och därför ej sorterar dem eller tar bort dubletter.

Format

En adress, ett radnummer eller både och kan finnas på alla typer av rader. Radnummer måste alltid skrivas längst till vänster på raden. En adress kan vara indragen och måste alltid avslutas med ett kolon (det får

inte vara något blanktecken mellan adress och kolon). Man kan blanda radnummer och adresser i ett program.

Konstanter

Det finns två typer av konstanter: sträng-konstanter och numeriska konstanter. En strängkonstant består av en följd av tecken inledda och avslutade med citationstecken (" ").

Exempel:

```
"abc123!£"
```

En strängkonstant kan innehålla upp till 32 767 tecken.

Numeriska konstanter är positiva eller negativa tal. Det finns sex olika typer av numeriska konstanter:

Korta heltal	Heltal mellan -32768 och +32767.
Långa heltal	Heltal mellan -2147483648 och +2147483647.
Decimaltal	Positiva och negativa reella tal, dvs tal som innehåller decimalpunkt och decimaler.
Flyttal	Positiva och negativa tal representerade i exponentform. Ett flyttal består av ett heltal eller decimaltal (bråkdelen) följt av bokstaven E och ett heltal (exponenten). Vid flyttal med dubbel precision så används bokstaven D istället för E.
Hexadecimala tal	Hexadecimala siffror med prefixet &H.
Oktala tal	Oktala siffror med prefixet &O eller &.

Decimaltal och flyttal kan vara i antingen enkel eller dubbel precision. Tal i enkel precision sparas med 7 siffrors noggrannhet (plus exponenten) och skrivs ut med 7 siffrors noggrannhet. Tal i dubbel precision sparas med 16 siffrors noggrannhet (plus exponenten) och skrivs ut med 16 siffrors noggrannhet. Se appendix D, "Amiga Basic:s interna representation av tal" för detaljer om hur tal lagras internt i datorn.

En numerisk konstant i enkel precision har någon av följande egenskaper:

* Högst sju siffror
* Exponentform markerat med ett E
* Ett avslutande utropstecken (!)

En konstant i dubbel precision har någon av följande egenskaper:

* Åtta eller fler siffror
* Exponentform markerat med ett D
* Ett avslutande brädgårdstecken (#)

Exempel på numeriska konstanter:

Enkel precision	**Dubbel precision**
46.8	345692811
-1.09E-6	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Numeriska konstanter får inte innehålla kommatecken (,).

Variabler

På samma sätt som för konstanter så finns det två typer av variabler: numeriska variabler och strängvariabler. Strängvariabler kan bara lagra strängar och numeriska variabler kan bara lagra tal. Innan man tilldelar ett värde till en variabel så innehåller den värdet noll (numeriska variabler) eller en tom sträng (strängvariabler).

Variabelnamn

Ett variabelnamn kan vara upp till 40 tecken långt. Man får använda bokstäver, siffror och punkt (.) i namnet men första tecknet måste vara en bokstav. Vissa specialtecken används också i variabelnamnet för att definiera olika typer av variabler, se "Typdefinition av variabler" i detta avsnitt.

Amiga Basic skiljer inte på små och stora bokstäver i variabelnamn.

Variabelnamnen ALPHA, Alpha och AlPhA refererar till samma variabel.

Om ett variabelnamn börjar med "FN" så antar Amiga Basic att det är ett anrop till en programdefinierad funktion. Se DEF FN i andra delen av detta kapitel för mer information om programdefinierade funktioner.

Reserverade ord

Med reserverade ord menas de ord som har en speciell funktion i Amiga Basic. Alla kommandon, instruktioner, funktioner och operatorer är reserverade ord, t ex GOTO, PRINT, TAN och AND. Reserverade ord bör alltid separeras från övriga tecken och ord i Amiga Basic (t ex variabler, datasatser) med blanktecken. Man kan inte använda reserverade ord som variabelnamn men reserverade ord får finnas inuti ett variabelnamn.

Exempel:

```
SANDPRIS
```

Här ingår det reserverade ordet "AND", men variabelnamnet är giltigt ändå.

Reserverade ord kan skrivas med både små och stora bokstäver. I appendix C, "Amiga Basic:s reserverade ord" finns en komplett uppräkning av de ord som är reserverade i Amiga Basic.

Typdefinition av variabler

Variabler kan definieras till att vara numeriska variabler eller till att vara strängvariabler.

Man avslutar en strängvariabels namn med ett dollartecken ($).

Exempel:

```
LET A$ = "SÄLJ-RAPPORT"
```

Dollartecknet är ett typdefinierande tecken, dvs tecknet definierar variabeln till att vara en strängvariabel.

Det finns flera olika typdefinierande tecken som används i samband med

numeriska variabler. Man kan definiera en numerisk variabel till att lagra korta eller långa heltal eller tal i enkel eller dubbel precision. Det passar oftast bäst att räkna med variabler med dubbel precision men de tar upp mer minne än variabler i enkel precision. Numeriska variabler är fördefinierade till enkel precision.

Här nedan visas de typdefinierande tecken som finns för variabler och det minnesutrymme (antal bytes, tecken) som variabler av respektive typ upptar.

%	Kort heltal	2
&	Långt heltal	4
!	Enkel precision	4
#	Dubbel precision	8
$	Sträng	5 bytes plus strängens innehåll

Man behöver inte använda typdefinierande tecken för att definiera en variabels typ. Man kan använda DEFINT-, DEFLNG-, DEFSTR-, DEFDBL- och DEFSNG-instruktionerna i ett program för att binda första bokstaven i ett variabelnamn till en speciell variabeltyp. Varje gång man sedan använder denna bokstav som första bokstav i ett variabelnamn så blir variabeln av den typ som bokstaven är bunden till. De här instruktionerna beskrivs noggrant under DEFINT i lathunden längre fram i det här kapitlet.

Indexerade variabler

En indexerad variabel består av ett antal variabler med samma variabelnamn men med olika index. Man kallar de enskilda variablerna för element. Elementen används precis på samma sätt som vanliga variabler med instruktioner och funktioner. Man definierar en indexerad variabels namn, typ och antal element som skall ingå.

Varje element i den indexerade variabeln identifieras av ett eller flera heltal. Antalet identifierande heltal som behövs för ett enskilt element är lika med antalet dimensioner som finns i den indexerade variabeln.

Exempel:

V(10) är ett element i en endimensionell variabel.

T(1,4) är ett element i en tvådimensionell variabel.

G(2,5,7) är ett element i en tredimensionell variabel.

Observera att den indexerade variabeln V(*n*) och den vanliga variabeln V inte är samma variabel.

Det maximala antalet dimensioner en indexerad variabel kan ha är 255 och det maximala antalet element per dimension är 32 768 stycken.

Enskilda element i indexerade strängvariabler behöver inte vara av samma längd.

Varje element i en indexerad variabel kräver lika mycket minnesutrymme som motsvarande vanlig variabel.

Konvertering av olika numeriska typer

Amiga Basic konverterar numeriska konstanter av en typ till en annan typ när det behövs. Följande regler gäller:

1. Om en numerisk variabel av en viss typ tilldelas värdet av en numerisk konstant av en annan typ, så sparas värdet i variabeln enligt variabelns och inte konstantens typ (om en strängvariabel tilldelas ett numeriskt värde eller vice versa, så får man ett "Type mismatch" felmeddelande).

2. Vid beräkningar av uttryck konverteras alla ingående operander till samma numeriska typ, vilket blir den typ som den mest precisa operanden har. Även resultatet av uttrycket får denna typ.

3. Logiska operatorer konverterar operanderna till heltal och returnerar ett heltal som resultat. Operanderna får inte vara större eller mindre än de tal som kan lagras som ett kort eller långt heltal.

4. När ett flyttal konverteras till ett heltal så avrundas decimaldelen.

Uttryck och operatorer

Ett uttryck är en kombination av konstanter, variabler, andra uttryck och operatorer. Uttrycket räknas ut av Basic-tolken och resulterar i en sträng eller ett numeriskt värde. Operatorerna utför matematiska eller logiska operationer på operandernas värden. Det finns fyra typer av operatorer i

Amiga Basic:

* Aritmetiska operatorer
* Relations operatorer
* Logiska operatorer
* Funktions operatorer

Olika operatorers prioritetsordning

Om det är flera operatorer i ett uttryck så kommer vissa operatorer att utföras före de andra, de har högre prioritetsordning. Om operatorerna har samma prioritetsordning så utförs de från vänster till höger. Följande prioritetsordning råder mellan operatorerna i Amiga Basic:

1. Exponentiering
2. Matematisk negation
3. Multiplikation och flyttalsdivision
4. Heltalsdivision
5. Modulo
6. Addition och subtraktion
7. Relationsoperatorer
8. NOT
9. AND
10. OR och XOR
11. EQV
12. IMP

Aritmetiska Operatorer

Nedan visas en lista på några aritmetiska operatorer i prioritetsordning:

Operator	**Operation**	**Exempel**
^	Exponentiering	X^Y
-	Matematisk negation	-X
*	Multiplikation	X*Y
/	Flyttalsdivision	X/Y
\	Heltalsdivision	X\Y
MOD	Modulo	Y MOD X
+, -	Addition, Subtraktion	X+Y,X-Y

Man kan ändra prioritetsordningen med hjälp av parenteser. Operationer inom parenteser utförs först. I parentesen gäller den vanliga prioritetsordningen.

Matematiska uttryck i Amiga Basic ser något annorlunda ut jämfört med sin algebraiska motsvarighet:

Algebraiskt uttryck	**Amiga Basic uttryck**
$\frac{X-Z}{Y}$	`(X-Z)/Y`
$\frac{XY}{Z}$	`X*Y/Z`
$\frac{X+Y}{Z}$	`(X+Y)/Z`
$(X^2)^Y$	`(X^2)^Y`
X^{Y^Z}	`X^(Y^Z)`
$X(-Y)$	`X*(-Y)`

Heltalsdivision

Vid heltalsdivision används omvänt snedstreck (\) istället för snedstreck (/) som används vid flyttalsdivision. Operanderna (täljaren och nämnaren) avrundas till heltal medan resultatet trunkeras (huggs av) till heltal.

Exempel:

```
X=10/4
Y=25.68\6.99
PRINT X,Y

2           3
```

Modulo

Modulo får man med operatorn MOD. Modulo ger den rest som uppstår vid heltalsdivision.

Exempel:

`10.3 MOD 4 = 2` (10\4 = 2 med resten 2)

`25.68 MOD 6.99 = 5` (26\7 = 3 med resten 5)

Observera att både täljare och nämnare avrundas till heltal före MOD-operationen utförs.

Division med noll och spill

Om nämnaren i ett uttryck blir 0 så kommer ett felfönster att öppnas med meddelandet "Division by zero" (engelska för "Division med noll"). När man klickat felfönstrets OK ruta så kommer det högsta tal som Amiga Basic kan framställa att visas på skärmen med täljarens tecken (+ eller -) framför sig. Sedan fortsätter körningen av programmet. Om noll upphöjs till ett negativt tal vid exponentiering så händer samma sak som ovan förutom att det alltid är ett positivt tal som visas.

Relationsoperatorer

Relationsoperatorer används för att jämföra två värden. Resultatet blir antingen sant (-1) eller falskt (0). Resultaten kan bl a användas för att styra programflödet (se IF...THEN...-instruktionen i lathunden). Följande

relationsoperatorer finns:

Operator	Relation	Exempel
=	Lika med, ekvivalens	X = Y
<>	Ej lika med, skilt från	X <> Y
<	Mindre än	X < Y
>	Större än	X > Y
<=	Mindre än eller lika med	X <= Y
>=	Större än eller lika med	X >= Y

Lika med symbolen (=) används också för att tilldela variabler värde, se LET i lathunden.

När aritmetiska operatorer och relationsoperatorer kombineras i uttryck så utförs de aritmetiska operationerna först, dvs de har högre prioritet. Prioriteten kan förstås ändras med parenteser.

Logiska operatorer

Med de logiska operatorerna kan man manipulera bitar, använda Boolsk algebra och testa flerställiga relationer. Logiska operatorer kan användas på samma sätt som relationsoperatorer.

En logisk operator returnerar värdet från ett uttryck med operatorer som har värdet sant eller falskt . Resultatet (i bitar) är antingen sant (-1) eller falskt (0). Det finns sex logiska operatorer i Amiga Basic:

NOT	Logiskt icke
AND	Konjunktion
OR	Disjunktion
XOR	Exklusiv disjunktion
IMP	Implikation
EQV	Ekvivalens

I följande *sanningstabell* visas vad varje operator returnerar för värde vid olika kombinationer av värden på operanderna. Operatorerna är uppställda i prioritetsordning.

NOT	X		NOT X
	T		F
	F		T
AND	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F
XOR	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F
IMP	X	Y	X IMP Y
	T	T	T
	T	F	F
	F	T	T
	F	F	T
EQV	X	Y	X EQV Y
	T	T	T
	T	F	F
	F	T	F
	F	F	T

Logiska operatorer omvandlar sina operander till tvåkomplements heltal med en förteckenbit (anger plus eller minus). Heltalets längd beror på om man definierat ett långt eller ett kort heltal.

Om operanderna är antingen 0 eller -1 så kommer operatorn att returnera värdet 0 eller -1. Operationen kommer att testas bitvis, dvs varje bit i resultatet bestäms av de korresponderande bitarna i

operanderna. Man kan därför använda logiska operatorer för att undersöka om en bitgrupp innehåller ett visst bitmönster. Man kan exempelvis använda AND för att maska fram en speciell bit. OR kan användas för att blanda två bitgrupper och skapa ett nytt binärt värde.

Exempel:

63 AND 16 = 16	63 är binärt 111111 och 16 är binärt 010000. 111111 AND 010000 = 010000.
15 AND 14 = 14	15 är binärt 1111 och 14 är binärt 1110. 1111 AND 1110 = 1110.
-1 AND 8 = 8	-1 är binärt 1111111111111111 och 8 är binärt 1000. 1111111111111111 AND 1000 = 1000
2 OR 4 = 6	2 är binärt 10 och 4 är binärt 100. 10 AND 100 = 110.
-1 OR -2 = -1	-2 är binärt 1111111111111110. -1finns ovan. 1111111111111111 OR 1111111111111110 = 1111111111111111.
NOT X = -(X+1)	Tvåkomplementet till ett heltal är bitkomplementet plus 1.

Funktioner och funktionsoperatorer

När man använder en funktion i ett uttryck så kallar funktionen på de operatorer som den är definierad av. Amiga Basic har två typer av funktioner: standardfunktioner som redan finns i systemet, t ex SQR (kvadratrot) och SIN (sinus), och programdefinierade funktioner som definieras av programmeraren. Se lathunden nedan för en exakt beskrivning av standardfunktioner och DEF FN.

Operationer på strängar

Ett stränguttryck består av strängkonstanter, strängvariabler och andra stränguttryck kombinerade av olika operatorer. Det finns tre olika typer av operationer man kan utföra på strängar: additions-, relations- och funktionsoperationer.

Addition av strängar

Man använder plus(+)-operatorn för att lägga ihop strängar.

Exempel:

```
LET A$ = "Fil" : LET B$ = "namn"
PRINT A$ + B$
PRINT "Nytt " + A$ + B$
END
```

Följande kommer upp på skärmen:

```
Filnamn
Nytt Filnamn
```

Jämförelse av strängar

När man jämför strängar kan man använda samma relationsoperatorer som när man jämför tal:

```
=   <   >   <>   <=   >=
```

När strängar testas mot varandra så tas ett tecken i taget från vänster i vardera strängen och deras ASCII-koder jämförs. I ASCII-koden har varje bokstav ett nummer. När dessa nummer jämförs så gäller samma regler som vid jämförelse av tal. Om man vid jämförelse av två strängar når slutet av den ena strängen och de båda strängarna dittills varit lika så sägs den kortare strängen vara mindre (värd) än den längre strängen.

Exempel på sanna uttryck:

```
"AA" < "BB"
"FILNAMN" = "FILNAMN"
"CL " > "CL"
"kg" > "KG"
"SVEN" < "SVENS"
```

Strängar måste vara omgivna av citationstecken när man jämför dem.

Amiga Basic: lathund

I detta avsnitt ges en fullständig beskrivning av alla instruktioner och funktioner i Amiga Basic (över 130 stycken!). De behandlas i bokstavsordning och presenteras på följande sätt:

Syntax	Korrekt och fullständig syntax för funktionen eller instruktionen visas. Syntaxen visas i rubriken för att den omedelbart skall gå att hitta.
Funktion	Beskrivning av vad funktionen eller instruktionen utför.
Användning	Beskrivning av parametrar i detalj, hur och till vad man använder instruktionen eller funktionen.
Se även	Referens till relaterade instruktioner eller funktioner.
Exempel	Förklarande exempel.
Anmärkning	Kommentar till viktiga egenskaper hos instruktionen eller funktionen.
Varning	Varning angående faror eller problem med användningen av instruktionen eller funktionen.

Följande notation används:

STORA BOKSTÄVER	Anger att parametern måste skrivas in som den står.
Kursiv text	Anger att användaren skall ange parametern.
[ ]	Parametrar inom hakparenteser är valfria.
	Anger att parametern kan upprepas hur många gånger som helst.
{}	Klamrar anger att man måste välja mellan två eller flera parametrar. En av dessa parametrar måste väljas om inte alla alternativen är omgivna av hakparenteser.

Lodräta streck åtskiljer parametrarna inom klamrar.

Tecken som komma, parenteser, semikolon, bindestreck och "lika med" måste användas där de visas i syntaxen.

ABS(X)

Returnerar absolutvärdet av X.

Exempel:

```
LET x=987 : LET y=-987
PRINT ABS(x), ABS(y)
```

Följande visas på skärmen:

```
987  987
```

AREA *[STEP](x,y)*

Definierar den punkt där en polygon skall ritas med AREAFILL-instruktionen.

Parametrarna *x* och *y* anger en av de punkter som skall sammanbindas för att bilda polygonen med AREAFILL-satsen. AREAFILL-instruktionen kan inte använda fler än 20 AREA-satser.

Om STEP används så räknas punkten *x,y* ut relativt till den grafiska pennans position i det aktiva fönstret. Annars anger koordinaterna en absolut punkt i fönstret.

Se även: AREAFILL

AREAFILL [läge]

Ändrar områdets utseende i en polygon definierad av två eller flera tidigare AREA-instruktioner.

läge anger utseendet för området:

0 Fyller området med det mönster som PATTERN anger. 0 är förvalt värde för *läge*.

1 Inverterar området.

Exempel:

Följande satser ritar upp en triangel och fyller det område som triangeln upptar:

```
AREA (10,10)
AREA STEP (0,40)
AREA STEP (40,-40)
AREAFILL
```

Se även: AREA, PATTERN och COLOR

ASC(X$)

Returnerar ASCII-koden för det första tecknet i X$.

I Amiga Basic:s teckenuppsättning ingår alla ASCII-tecken och en del andra tecken. Alla tecken kan testas med ASC-funktionen. Se appendix A, "Teckenkoder".

Se även: CHR$

Exempel:

```
LET Object$ = "T"
PRINT ASC(Object$)
END
```

Följande skrivs ut på skärmen:

```
84
```

ATN(X)

Returnerar arctangens för X. X skall vara i radianer. Resultatet är inom området -pi/2 till pi/2 radianer.

Funktionen räknar med och returnerar talen i samma precision som X har.

Exempel:

```
'Arctangent request program
newnumber:
INPUT "Enter a number ", NUMBER
PRINT "Arc tangent of "  NUMBER " is " ATN(NUMBER)
INPUT "If you have another number, enter y ", YORN$
IF YORN$ = "y" GOTO newnumber
END
```

Resultatet av programmet skulle kunna bli:

```
Enter a number 33
Arc tangent of 33 is 1.540503
If you have another number enter y y
Enter a number 2
Arc tangent of 2 is 1.107149
If you have another number, enter y n
```

BEEP

Avger en ljudsignal och skärmen blinkar en gång.

Exempel:

```
IF Minnekvar& < 100 THEN
   BEEP
   LOCATE 17,1
   PRINT "SLUT PÅ MINNET! MINSKA BILDENS STORLEK";
END IF
```

BREAK {ON}{OFF}{STOP}

Sätter på (ON), stänger av (OFF) och avbryter tillfälligt (STOP) avbrottshanteringen av användarens försök att avbryta programkörningen.

Vid BREAK STOP så kommer inte ON BREAK...GOSUB-satsen att utföras. Dock kommer användarens försök att avbryta programmet lagras så att så fort en BREAK ON-sats utförs så kommer ON BREAK...GOSUB-satsen att utföras om användaren försökt stoppa programmet under den tid avbrottshanteringen tillfälligt var avbruten.

Se även: ON BREAK

Exempel:

```
BREAK ON
ON BREAK GOSUB DIRECTUSER
DIM PAYTIME(99),HRS(99),GROSS(99),FIT(99),FICA(99),STATE(99),NET(99)
LET TOTALEMPLOYEES = 99
OPEN "O",#1,"EmployeePay"
   FOR I=1 TO TOTALEMPLOYEES

WRITE#1,PAYTIME(I),HRS(I),GOSS(I),FIT(I),FICA(I),STATE(I),NET(I)
    NEXT I
CLOSE #1 :BREAK OFF
INPUT "Do you wish to print the payroll now (Y/N)?", ANSWER$
IF ANSWER$ = "Y" THEN BREAK ON: GOSUB PRINTCHECKS
END
DIRECTUSER:
   CLS:BEEP:PRINT "You can't exit program until file is updated."
   RETURN
```

CALL namn *[(parameterlista)]* (1)
namn [parameterlista] (2)

CALL används när man vill göra:

1. Anrop till subprogram definierade av SUB-instruktioner.

2. Anrop till maskinspråksrutiner på fast adress.

3. Anrop till en biblioteksrutin (skriven i maskinspråk).

Själva ordet CALL är valfritt; om man utesluter CALL så

skall även parenteserna omkring parameterlistan tas bort, som vid syntax 2 ovan.

Anrop av underprogram

Vid anrop av underprogram kan variabelvärden överföras med hjälp av variabelreferens medan uttryck överförs som värde.

Exempel:

```
SUB Alpha (x,y) STATIC
END SUB
CALL Alpha (a,b)
```

Se även: SUB (i detta kapitel och i kapitel 6)

Anrop av maskinspråksrutiner

CALL-instruktionen är det enda sättet som finns att anropa en extern rutin. "namn" är då namnet på den enkla variabel som innehåller startadressen i minnet för maskinspråksrutinen. Variabeln får inte vara ett element i en indexerad variabel.

Parametrarna i parameterlistan överförs som värde i enlighet med den standard som finns för programspråket C. Alla parametrar måste vara korta eller långa heltal, annars så får man ett "Type mismatch" felmeddelande. Adressen till en variabel i enkel eller dubbel precision kan skickas till maskinspråksrutinen på följande sätt:

```
CALL Rutin(VARPTR(x))
```

Adressen till en strängvariabel kan skickas med:

```
CALL Rutin(SADD(x$))
```

I följande exempel är variabeln som innehåller minnesadressen definierad som ett kort heltal (&). Använd långa heltalsvariabler om adresslängden är 24 bitar då en kort heltalsvariabel eller en variabel i enkel precision inte kan lagra en 24 bitars adress.

Exempel:

```
a=0 : b=0
DIM Kod%(100)
FOR i=0 TO 90
  READ Kod%(I)
NEXT i
Kodadr&(a,b)=VARPTR(Kod%(0))
CALL Kodadr(a,b)
```

Anrop av maskinspråksrutiner från ett bibliotek

Biblioteksrutiner är maskinspråksrutiner som alltid finns tillgängliga när man kör ett Amiga Basic program. De är speciella källfiler i Amiga systemet.

Parametrar överförs som värden i enlighet med programspråket C.

Exempel:

```
LIBRARY graphics.library
CALL Draw(50,60)
```

Amiga Basic skapar en variabel med namnet Draw som kommer att innehålla adressen till maskinspråksrutinen. Draw kan därför inte vara en heltalsvariabel. Till exempel så skulle:

```
DEFINT A-Z
CALL Draw(50,60)
```

ge ett "Type mismatch" felmeddelande medan:

```
DEFINT A-Z
CALL Draw#(50,60)
```

skulle fungera utmärkt.

Observera att det typdeklarerande tecknet (#) ignoreras när biblioteket genomsökes efter rutinen. I exemplet ovan så kommer Amiga Basic att söka efter Draw i biblioteket och inte efter Draw#.

Varning:

Då ordet CALL kan utelämnas så kan man anropa en rutin med syntaxen:

namn parameterlista

Ett sådant anrop liknar mycket en alfanumerisk adress.

Exempel:

```
Alpha: LET a=5
```

I ovanstående exempel kan man inte se om satsen är ett anrop på ett underprogram med namnet Alpha utan någon parameterlista, eller om det är en rad med adressen Alpha innehållande satsen LET a=5. I sådana här fall antar Amiga Basic att det är en radadress och inte ett underprogramanrop utan parameterlista.

Efter satserna THEN och ELSE så måste ordet CALL vara med för att skilja identifieraren från en adress.

CDBL(X)

Konverterar X till ett tal i dubbel precision.

Exempel:

```
a!=6666 : b!=100000!
PRINT a!*b!, "(resultatet utskrivet i enkel precision)"
PRINT CDBL(a!*b!), "(resultatet utskrivet i dubbel precision)"
```

På skärmen visas:

```
6.666E+08    (resultatet utskrivet i enkel precision)
666600000    (resultatet utskrivet i dubbel precision)
```

CHAIN [MERGE] *filnamn*[,*[uttryck]* [ALL] [,DELETE *område*]]

CHAIN läser in och kör ett program och överför variabelvärden från programmet i minnet till det nya programmet.

filnamn är namnet på programmet som anropas.

uttryck är en radadress, ett radnummer eller ett uttryck som ger ett

radnummer som finns i det anropade programmet. Därifrån kommer programmet att starta. Om ingen startpunkt anges så börjar programkörningen från första raden.

MERGE gör att man kan addera programrader (oftast subrutiner) till programmet. Raderna läggs i slutet på programmet i minnet. Programraderna måste vara lagrade som en ASCII-fil.

ALL gör att alla variabler, utom de som är lokala i ett underprogram, överförs till det anropade programmet. Om ALL inte används så måste programmet i minnet innehålla en COMMON-instruktion som räknar upp de variabler som skall överföras till det anropade programmet.

Om ALL används men inte *uttryck*, så måste ett kommatecken skrivas på den plats som *uttryck* annars skulle ha upptagit.

CHAIN lämnar filer öppna.

Om man adderat ett antal programrader, t ex en subrutin, och sedan inte längre behöver dem så kan det vara lämpligt att ta bort raderna för att få utrymme att addera en ny programdel. Använd DELETE-optioner för att ta bort programrader.

område består av ett radnummer eller av en radadress, ett bindestreck, och ett nytt radnummer eller en ny radadress. Alla rader mellan de två angivna raderna, inklusive dem själva, tas bort.

Anmärkning:

CHAIN-instruktionen tillsammans med optionen MERGE bevarar aktuell OPTION BASE (se nedan) inställning.

Om MERGE inte används så kommer inte definitioner av variabeltyper och egendefinierade funktioner att bevaras så att det anropade programmet kan använda dem. Det vill säga att alla DEFINT-, DEFLNG-, DEFSNG-, DEFSTR-, DEFDBL- och DEF FN-instruktioner måste utföras om igen i det anropade programmet. Avbrottshanteringen stängs också av så att om man vill ha avbrottshantering i det anropade programmet måste den sättas på igen.

När man använder MERGE och DELETE så skall man tänka på att placera egendefinierade funktioner före det område som man specificerat i DELETE annars så kommer funktionerna inte att bevaras.

Se även: COMMON, MERGE

Exempel:

```
COMMON Ackresult, Balans!, Krav(), Diskonto!, Kontrakt$
CHAIN "Reserver"
```

CHR$(I)

Returnerar en sträng som består av det tecken som har ASCII-värdet I. Se appendix A, "Teckenkoder".

Man använder oftast CHR$ för att skicka specialtecken till skärmen eller till annan utrustning. Man kan exempelvis skicka beep-tecknet, CHR$(7), till skärmen vilket ger samma effekt som BEEP-instruktionen. Sidframmatningstecknet CHR$(12), rensar resultatfönstret och ställer textmarkören i övre vänstra hörnet om det skickas till skärmen.

Exempel:

```
CLS
FOR I = 65 TO 90
PRINT CHR$(I); SPC(1);
NEXT I
```

Följande visas på skärmen:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

CHDIR *sträng*

Byter aktuell katalog.

sträng är namnet på katalogen.

Exempel:

```
CHDIR "df1:"    'rotkatalogen på skivenhet 1 blir aktuell katalog
CHDIR "df0:c"   'katalog c på skivenhet 0 blir aktuell katalog
CHDIR "/"       'moderkatalogen blir aktuell katalog
```

CINT(X)

Omvandlar X till ett heltal genom att runda av decimaldelen.

Om X inte är inom talområdet -32768 och 32767 så kommer man att få ett "Overflow" felmeddelande.

Anmärkning: Om X är ett tal som ligger mittemellan två heltal (som t ex 14.5) och heltalsdelen av X är ett jämnt tal så kommer X att avrundas nedåt; om heltalsdelen är ett ojämnt tal så rundas det uppåt.

Exempel:

```
PRINT CINT(-3.5)
PRINT CINT(-3.2)
FOR I = 1 TO 3
X = RND*10
PRINT X, "= random number generated by RND, times 10"
PRINT CINT(X), "= integer portion of the same number"
NEXT I
```

Följande visas på skärmen:

```
-4
-3
1.213501    = random number generated by RND, times 10
1           = integer portion of the same number
6.518611    = random number generated by RND, times 10
7           = integer portion of the same number
8.686811    = random number generated by RND, times 10
9           = integer portion of the same number
```

Se även: CLNG, CDBL, CSNG, FIX, INT

CIRCLE [STEP]*(x,y),radie* *[,färgid [,start,slut [,höjd/bredd-förhållande]]]*

Ritar upp en cirkel med given radie och centrumpunkt eller en ellips med given radie, centrumpunkt och höjd/bredd- förhållande.

x och *y* parametrarna anger x och y koordinaterna för cirkeln/ ellipsens

centrumpunkt.

STEP gör att centrumpunkten räknas ut relativt till pennans position. Om pennan befinner sig vid punkten 10,10 så kommer CIRCLE STEP(20,15) att referera till punkten 30,25.

radie anger cirkelns radie räknat i bildelement.

färgid anger vilken färg som skall användas. Den är relaterad till *färgid* i en PALETTE-sats. Förvalt värde är den aktuella förgrundsfärgen som den är definierad av COLOR-satsen.

Med *start-* och *slut*-parametrarna kan man bestämma vid vilken radian som cirkeln skall börja ritas och var den skall sluta. Det gör att man kan rita delar av en cirkel, t ex halvcirklar. *start* och *slut* kan variera mellan -2pi och 2pi. Om någon av parametrarna är negativ så kommer cirkelns kant att förbindas med en linje till centrumpunkten men parameterns värde kommer att behandlas som om det vore positivt. *start* kan vara mindre än *slut.*

höjd/bredd-förhållandet avgör om den uppritade figuren blir en cirkel eller en ellips. Skärmens inbyggda höjd/bredd-förhållande varierar mellan olika märken. Om *höjd/bredd-förhållandet* sätts till den använda skärmens höjd/bredd-förhållande så kommer en perfekt cirkel att ritas upp. En standard Amiga-skärm har vid högupplösning (640*200 bildelement) ett höjd/bredd-förhållande på 1/2.25 (approximativt 0.44) vilket är det förvalda värdet för *höjd/bredd- förhållandet.* Om man utelämnar *höjd/bredd-förhållandet* eller sätter det till 0.44 så kommer en perfekt cirkel att ritas på Amiga-skärmen.

Exempel 1:

```
CIRCLE (60,60),55
```

Ovanstående sats ritar en cirkel med radien 55 och med centrum i punkten 60,60.

Exempel 2:

```
Höjdbredd=0.1
WHILE Höjdbredd<20
   CIRCLE(60,60),55,0,,,Höjdbredd
   Höjdbredd=Höjdbredd*1.4
WEND
```

Ovanstående program ritar ett antal ellipser med varierande höjd/bredd-förhållande. 0 anger färgen som kommer att bli Amigans blå bakgrundsfärg om man inte specificerat annat i en PALETTE-sats.

CLEAR *[,[Amiga Basic-minnet][,stackminnet]]*

Raderar alla variabler och fördelar minnet mellan stacken, Amiga Basic- och systemminnet. Alla filer stängs och alla definitioner: DEF FN-, DEFINT-, DEFLNG-, DEFSNG-, DEFDBL- och DEFSTR-satserna nollställs.

Amiga Basic-minnet är ett numeriskt värde större än eller lika med 1024 som anger hur mycket minne som skall tilldelas för programkod, variabler, strängar och filbuffert. Om parametern utelämnas så kommer Amiga Basic-minnet att behålla sin aktuella storlek.

stack är ett numeriskt värde större än eller lika med 1024 som anger hur mycket minne som skall tilldelas stackminnet. Om parametern utelämnas används aktuellt värde.

Se även: FRE

Exempel:

```
CLEAR
CLEAR 20000
CLEAR , ,2000
CLEAR,20000,2000
```

CLNG*(numeriskt uttryck)*

Omvandlar ett numeriskt uttryck till ett långt heltal, eventuell decimaldel avrundas.

CLOSE *[[#]filnummer[,[#]filnummer ...]]*

Stänger filer för I/O.

CLOSE utan parametrar stänger alla filer.

filnummer är det nummer som filen blev öppnad under. CLOSE tar bort sambandet mellan en fil och *filnummer*. Efter att CLOSE-satsen utförts kan man knyta filnumret till en ny fil.

När man stänger en sekventiell fil med CLOSE så skrivs filbuffertens innehåll först till filen, därefter stängs den.

Se även: CLEAR, END, NEW, OPEN, RESET, STOP, SYSTEM

Exempel:

```
ACCOUNT=4
OPEN "Payables" AS #2 LEN = 80
  FIELD #2, 30 AS FIRM$, 30 AS ADDR$, 10 AS OWE$, 10 AS DAY$
  GET #2, ACCOUNT
          LET DEBT! = CVS(OWE$)
          LET DEBT! = DEBT! + CHARGES! - PAID
          LSET OWE$ = MKS$(DEBT!)
  PUT #2, ACCOUNT
CLOSE #2
PRINT "Account #";ACCOUNT;" updated"
```

Programmet öppnar en befintlig fil, läser data från filen, uppdaterar datat och skriver tillbaka det till filen.

CLS

Raderar aktuellt resultatfönster och sätter pennans position till övre vänstra hörnet i samma resultatfönster.

Exempel:

```
CLS
```

COLLISION*(objektid)*

Ger ett värde som anger vilken typ av kollision som inträffat.

Amiga Basic kan hålla upp till 16 inträffade kollisioner i minnet. Om denna gräns överskrids kommer de efterföljande kollisionerna inte att

lagras. Varje gång en kollision rapporteras till programmet, dvs då programmet utför en COLLISION-sats, blir en plats ledig i minnet för att lagra en ny kollision.

objektid refererar till *objektid* i en OBJECT.SHAPE-sats och anger vilket objekt som skall testas. *objektid* är ett numeriskt värde mellan -1 och n. Om *objektid* är 0 så returnerar funktionen identifikationsnumret för ett objekt som kolliderar med ett annat objekt, utan att ta bort någon information från minnet. Om *objektid* är -1 så returnerar funktionen identifikationsnumret för fönstret i vilken kollisionen identifierad av COLLISION(0) uppkom.

Om *objektid* är ett positivt tal (förutom 0) så returnerar funktionen identifikationsnumret för det objekt som kolliderat med objektet identifierat av *objektid.* Informationen tas bort från minnet.

Om funktionen returnerar ett negativt nummer mellan -1 och -4 så har objektet identifierat av *objektid* kolliderat med någon av de fyra fönsterlisterna:

-1 övre list

-2 vänster list

-3 nedre list

-4 höger list

Se även: OBJECT.SHAPE (som innehåller ett förklarande exempel)

COLLISION {ON}{OFF}{STOP}

Sätter på (ON), stänger av (OFF), och avbryter tillfälligt (STOP) avbrottshanteringen som reagerar på objektkollisioner. En objektkollision uppkommer då ett objekt definierat av en OBJECT.SHAPE-sats kolliderar med ett annat objekt eller med en fönsterlist.

En ON COLLISION...GOSUB-sats utförs då en kollision inträffar om COLLISION är ON.

Vid COLLISION STOP så kommer inte ON COLLISION...GOSUB-satsen att utföras. Dock kommer eventuella kollisioner att lagras. Så fort en

COLLISION ON-sats utförs så kommer ON COLLISION...GOSUB-satsen att utföras om en kollision inträffat under den tid avbrottshanteringen tillfälligt var avbruten.

Se även: COLLISION, OBJECT.SHAPE (även ett exempel), "Avbrottshantering" i kapitel 6, "Avancerad programmering"

COLOR *[förgrundsfärgid] [,bakgrundsfärgid]*

Anger vilken förgrundsfärg och bakgrundsfärg som skall användas.

Förgrundsfärgen används när områden skall fyllas med en viss färg, när linjer och punkter skall ritas, text skall färgas etc. Bakgrundsfärgen används för att omge dessa objekt.

förgrundsfärgid och *bakgrundsfärgid* är relaterade till *färgid* i en PALETTE-sats eller till den förvalda färgidentifieraren i Amiga systemet.

Om inga COLOR- och PALETTE-instruktioner används så kommer Amiga Basic att använda sig av Workbenchens färger. Dessa är förvalda till vit förgrund och blå bakgrund men kan ändras av användaren med Preferences-verktyget.

Se även: PALETTE

Exempel:

```
PALETTE 1,RND,RND,RND
PALETTE 2,RND,RND,RND
COLOR 1,2
```

COMMON *variabellista*

Skickar variabler till ett annat program som läses in med CHAIN.

COMMON-satser kan skrivas var som helst i programmet men det är god programmeringsvana att skriva dem i början. Programmet blir då lättare att förstå och risken minskar att CHAIN utförs före COMMON-satsen.

Samma variabler får förekomma i flera COMMON-satser.

Indexerade variabler specificeras genom att parenteser (dvs "()") läggs till i slutet av variabelnamnet.

Om alla variabler skall skickas, använd CHAIN tillsammans med ALL och utelämna COMMON-satsen.

Exempel:

```
COMMON Ackresult, Balans!, Krav(), Diskonto!, Kontrakt$
CHAIN "Reserver"
```

CONT

Återupptar programkörning då ett program blivit avbrutet av en STOP-instruktion, ett STOP-kommando eller av Amiga-punkt (.). CONT kan också användas för att återgå till normal programkörning vid stegvis körning.

Programkörningen återupptas på samma ställe som den blev avbruten. Om avbrottet kom när en INPUT-sats kördes så kommer körningen att återupptas med att INPUT-satsens utskrift ("?" eller ledtexten som angivits i INPUT-satsen) skrivs på skärmen.

Vid felsökning kombinerar man ofta CONT med STOP-instruktionen. När programmet är avbrutet kan man använda sig av det interaktiva läget i resultatfönstret för att kontrollera olika variablers värden och eventuellt ändra dessa.

Man kan också använda GOTO som kommando i resultatfönstret för att återuppta programkörningen på en specificerad rad.

Man kan återuppta programkörningen efter att ett programfel avbrutit körningen.

Man kan inte återuppta programkörningen om man ändrat i programtexten under avbrottet.

Se även: GOTO, STOP

```
CHECK!=25 : DEBIT!=9.89
PRINT CHECK!,DEBIT!
        STOP
BALANS!=CHECK!-DEBIT!
PRINT BALANS
END
```

COS(X)

Ger cosinus av X. X skall vara i radianer.

Beräkningen sker i enkel eller dubbel precision beroende på om X är i enkel eller dubbel precision.

Exempel:

```
PRINT "COSINUS AV 1 ÄR " COS(1)
PRINT "COSINUS AV 100 ÄR " COS(100)
PRINT "COSINUS AV 1000 ÄR " COS(1000)
```

Följande visas på skärmen:

```
COSINUS AV 1 ÄR .5403023
COSINUS AV 100 ÄR .8623189
COSINUS AV 1000 ÄR .5623791
```

CSNG(X)

Omvandlar det numeriska uttrycket X till ett tal i enkel precision.

Se även: CDBL, CINT

Exempel:

```
A#=6666 : B#=100000
PRINT A#*B#, "(resultatet i dubbel precision)"
PRINT CSNG(A#*B#), "(resultatet i enkel precision)"
```

Följande visas på skärmen:

```
666600000     (resultatet i dubbel precision)
6.66E+08      (resultatet i enkel precision)
```

CSRLIN

Ger approximativt det radnummer som pennan befinner sig på, relativt till det aktuella resultatfönstrets övre list.

Värdet är lika med eller större än ett.

För att bestämma radnumret använder CSRLIN tecknet "0":s höjd. Höjden bestäms av vilken fontdefinition som gäller i det aktuella resultatfönstret. Denna höjd är större än eller lika med 1.

Se även: POS, LOCATE

Exempel:

```
Y = CSRLIN  ' GET CURRENT CURSOR LINE NUMBER (VERTICAL POSITION)
X = POS(0)  ' GET CURRENT CURSOR COLUMN NUMBER (HORIZONTAL POSITION)
LOCATE 20,1 ' PLACE CURSOR ON LINE 20, COLUMN 1 (BOTTOM OF SCREEN)
PRINT "THIS PRINTS AT LOCATION 20,1 (BOTTOM OF PAGE)"
LOCATE Y,X  ' PLACE CURSOR IN ORIGINAL LOCATION
PRINT "THIS PRINTS AT ORIGINAL LOCATION OF CURSOR"
```

Exemplet ovan lagrar markörens aktuella position, flyttar markören längst ner på skärmen, skriver ett meddelande, flyttar tillbaka markören till originalpositionen och skriver ett meddelande även där.

CVI*(2-bytes-sträng)*
CVL*(4-bytes-sträng)*
CVS*(4-bytes-sträng)*
CVD*(8-bytes-sträng)*

Omvandlar en numerisk sträng som lästs från en direktåtkomstfil till ett numeriskt värde.

CVI omvandlar en 2-bytes-sträng till ett kort heltal.

CVL omvandlar en 4-bytes-sträng till ett långt heltal.

CVS omvandlar en 4-bytes-sträng till ett tal i enkel precision.

CVD omvandlar en 8-bytes-sträng till ett tal i dubbel precision.

CVI, CVL, CVS, CVD används med FIELD- och GET-satser för att omvandla numeriska strängar, som läses från en direktåtkomstfil, till tal. För att i andra fall få det numeriska värdet ur en sträng, använd VAL-funktionen.

Se även: MKI$, MKL$, MKS$, MKD$, VAL

Exempel:

```
OPEN Filnamn$ FOR INPUT AS 1
Färgset=CVL(INPUT$(4,1))
Dataset=CVL(INPUT$(4,1))
```

DATA *lista på konstanter*

Lagrar numeriska konstanter och strängkonstanter. Dessa kan sedan läsas med READ-instruktionen.

DATA-satser är icke-körbara satser och kan därför placeras var som helst i ett program.

DATA-satsen kan innehålla så många konstanter som får rum på en programrad. Konstanterna måste åtskiljas av kommatecken. Man kan använda så många DATA-satser som Amiga Basic-minnet klarar av att lagra.

READ-instruktionen läser DATA-satserna i den ordning de kommer i programmet, uppifrån och ned. Man kan se alla DATA-satser som en sammanhängande lista även om de råkar ligga på olika ställen i programtexten.

lista på konstanter kan innehålla vilken typ av numeriska värden som helst men inte uttryck.

Strängkonstanter måste omges med citationstecken om de innehåller kommatecken, kolon eller mellanslag. I andra fall behövs inte

citationstecknen.

Variabeltyperna i READ-satsen måste överensstämma med korresponderande konstanter i DATA-satserna. DATA-satserna kan läsas från början med hjälp av RESTORE-instruktionen.

Se även: READ, RESTORE

Exempel:

```
DIM Pattern0%(3)
DIM Pattern1%(3)
DIM Pattern2%(3)
FOR I=0 TO 3
  READ Pattern0%(I)
  READ Pattern1%(I)
  READ Pattern2%(I)
NEXT I
DATA &HAAAA, &H3333, &HFFFF
DATA &H5555, &H3333, &HFFFF
DATA &HAAAA, &H3333, &HFFFF
DATA &H5555, &H3333, &HFFFF
```

DATE$

Ger systemets datum på formen *mm-dd-åååå*

Exempel:

```
10 PRINT DATE$      'Skriv systemdatum
```

DECLARE FUNCTION *id [(parameterlista)]* LIBRARY

Gör att Amiga Basic söker genom de bibliotek som är öppnade med en LIBRARY-instruktion efter maskinspråksfunktionen *id* när den påträffas i ett uttryck.

Se LIBRARY för mer information om hur man öppnar ett bibliotek.

id är en identifierare som kan innehålla ett typdeklarerande tecken: %, &, !, #. *id* anger vilken maskinspråksfunktion som avses och vilken typ det skall vara på värdet som funktionen returnerar.

parameterlista är en lista på de parametrar som funktionen behöver. Denna lista behövs inte för funktionens skull men kan vara bra för att dokumentera funktionens ändamål.

Om funktionen existerar så skickas alla parametrar (om det finns några) till funktionen. *id*:s typdefinierande tecken (om det finns några) bestämmer vilken typ värdet som funktionen returnerar blir. T ex så returnerar Alpha# ett värde i dubbel precision, Beta% returnerar ett heltalsvärde. Om *id* inte har något typdefinierande tecken så gäller standardreglerna för typdefinition (se DEFINT).

Se CALL för en beskrivning av de regler som gäller för överföring av parametrar.

Se även: CSNG, DEFINT, DEFSNG, LIBRARY

Exempel:

```
DECLARE FUNCTION WiewPortAddress&() LIBRARY
LIBRARY "intuition.library"
VPA&=WiewPortAddress&(WINDOW(7))
```

Värdet som funktionen WiewPortAddress& returnerar kommer att tilldelas variabeln VPA&.

DEF FN*namn[(lista på formella parametrar)] =funktionsdefinition*

Definierar en funktion.

namn måste vara ett giltigt variabelnamn. Det får inte vara några mellanslag mellan DEF FN och *namn*. När *namn* påträffas i programmet kommer den definierade funktionen att utföras.

lista på formella parametrar innehåller de variabler som finns i *funktionsdefinitionen*. Varje variabel måste åtskiljas av ett kommatecken. Dessa variabler kommer att få sina värden genom referens till de aktuella parametrar som finns i funktionsanropen.

funktionsdefinitionen är ett uttryck som beskriver vad för operation funktionen skall utföra. Längden är begränsad till en rad. Variabelnamnen i uttrycket påverkar inte variabler i programmet med

samma namn.

När en funktion utförs så kommer en variabel specificerad både i den formella parameterlistan och i funktionsdefinitionen att innehålla samma värde. Annars så kommer det aktuella värdet, dvs det värde som variabeln eventuellt tidigare tilldelats, att användas.

Man kan skapa både numeriska funktioner och strängfunktioner med DEF FN. Funktionen returnerar värden av den typ som var specificerad i anropsatsen. Om datatypen i anropet inte passar datatypen i funktionen så får man ett "Type mismatch" felmeddelande.

Anmärkning: Om man har två funktionsdefinitioner med samma namn kommer den senast definierade funktionen att gälla.

DEF FN måste placeras före anropet annars får man felmeddelandet "Undefined user function". Man kan inte definiera en funktion i interaktivt läge eller i ett subprogram.

En funktionsdefinition gäller bara i det program där DEF FN finns. Om ett annat program anropas med CHAIN så kommer inte funktionsdefinitionerna i det gamla programmet att gälla i det nya programmet.

Exempel:

```
DEF FNPERCENT(A,B) = (A/B)*100
INPUT "ENTER PORTION OF TOTAL AMOUNT ", PORTION
INPUT "ENTER THE TOTAL ", TOTAL
RESULT = FNPERCENT(PORTION,TOTAL)
PRINT "PERCENTAGE IS ";RESULT;"%"
```

Följande är ett exempel på vad som skulle kunna visas på skärmen om ovanstående program körs.

```
ENTER PORTION OF TOTAL AMOUNT 276
ENTER THE TOTAL 1000
PERCENTAGE IS 27.6 %
```

DEFINT *bokstavsområde*
DEFLNG *bokstavsområde*
DEFSNG *bokstavsområde*
DEFDBL *bokstavsområde*
DEFSTR *bokstavsområde*

Gör att första bokstaven i ett variabelnamn kopplas till en viss variabeltyp. Följande typer kan definieras:

Instruktion	Variabeltyp	Typdeklarerande tecken
DEFINT	Kort heltal	%
DEFLNG	Långt heltal	&
DEFSNG	Enkel precision	!
DEFDBL	Dubbel precision	#
DEFSTR	Sträng	$

Man kan övertrumfa DEFINT, DEFLNG, DEFSNG, DEFDBL och DEFSTR med de typdefinierande tecknen %, &, !, #, $.

Exempel:

```
DEFLNG a-p,w
```

Ovanstående sats gör att alla variabler som börjar på en bokstav från a till p och w kommer att bli definierade till heltal.

DELETE *[rad][-rad]*

Tar bort programrader.

DELETE kan avse rader med både radnummer och radadresser. Om *rad* inte existerar får man felmeddelandet "Illegal function call".

DIM [SHARED] *variabellista*

Definierar storleken på variabler och reserverar minne för dem.

Använd DIM när indexet hos en variabel skall överskrida 10, annars får man felmeddelandet "Subscript out of range" ("index för stort"). Det minsta värdet för indexet är 0 om man inte definierat om värdet med OPTION.BASE.

DIM sätter alla element i den indexerade variabeln till noll (numeriska element) eller tom sträng (strängelement).

Det teoretiskt maximala antalet dimensioner i en DIM-sats är 255, men i praktiken är det minnesutrymmet som bestämmer hur många man kan ha.

SHARED gör variablerna globalt delade, dvs man kan använda variablerna även i underprogram. Genom DIM SHARED slipper man ha likadana SHARED-satser i flera underprogram.

Om en variabel har definierats eller använts och sedan används i en DIM-sats så får man felmeddelandet "Redimension array" ("Omdefiniering av indexerad variabel"). Placera DIM-satserna i början av programmet så att de utförs innan variablerna ifråga används.

Se även: SHARED

Exempel:

```
DIM SHARED A,B,C(10,2)
DIM CF(19)
FOR I=1 TO 19
  READ CF(I)
  PRINT CF(I)
NEXT I
 DATA 0,2,4,5,7,9,11,0,1,-1, 0,0,0,0,0,0, -12,12,0
```

END

Avslutar programmet, stänger alla filer och återvänder till föregående arbetsläge.

END kan placeras var som helst i ett program. Man behöver inte ha en

END-sats i slutet av ett program.

EOF*(filnumer)*

Testar om slutet på en fil nåtts.

Funktionen returnerar värdet -1 (sant) om slutet har nåtts på en sekventiell fil. Genom att använda EOF() för att undersöka om slutet på filen nåtts när man läser in data med INPUT# så kan man undvika felet "Input past end" ("Läsning förbi filslut").

När man använder EOF tillsammans med direktåtkomstfiler så returnerar funktionen värdet sant om den senaste GET-satsen inte kunde läsa en hel post. Detta är detsamma som att GET-satsen försökte läsa förbi filslutet.

Exempel:

```
OPEN "I",#1,"INFO"
   LINE INPUT #1, LONG$
   PRINT LONG$
CLOSE #1
OPEN "I",#1, "INFO"
   WHILE NOT EOF(1)
      PRINT ASC(INPUT$(1,#1));
      LET C = C + 1: IF C = 10 THEN PRINT: LET C = 0
    WEND
CLOSE #1
END
```

ERASE *lista på indexerade variabler*

Raderar indexerade variabler från minnet så att man får mer ledigt minnesutrymme.

Man kan definiera om en indexerad variabel sedan den blivit raderad. Man kan inte definiera om en tidigare definierad och ej raderad indexerad variabel.

Exempel:

```
ERASE Matris
```

ERR
ERL

Ger felnummer (ERR) respektive radnumret (ERL) på den raden som ett fel har uppkommit på.

Om den rad som felet uppkom på inte har något radnummer kommer ERL att returnera radnumret på den första numrerade raden före raden. ERL returnerar inte radadresser.

ERR och ERL används ofta i IF...THEN satser för att anropa en felhanteringsrutin.

Om ett fel uppkommer då man matar in en sats i interaktivt läge så returnerar ERL värdet 65535

I appendix B, "Felkoder och felmeddelande" finns en komplett uppräkning av alla Amiga Basics felkoder med förklaringar av eventuella orsaker.

Exempel:

```
ON ERROR GOTO Fixafel
     .
     .
     .
Fixafel:
  IF (ERR=55) AND (ERL=90) THEN CLOSE : RESUME
```

ERROR *heltalsuttryck*

Med ERROR kan man simulera Amiga Basic-fel eller definiera egna felkoder.

ERROR kan användas både som ett interaktivt kommando och som en instruktion i ett program.

heltalsuttryck är större än 0 och mindre än 256. Om *heltalsuttryck* är en felkod som Amiga Basic använder så kommer motsvarande felmeddelande att skrivas ut. Om avbrottshantering av fel är på så skrivs ingenting ut.

När man definierar sina egna felkoder använder man ett värde som är högre än det högsta som Amiga Basic använder. Det är lämpligt att använda ett så stort värde som möjligt för att undvika att krocka med de koder framtida versioner av Amiga Basic kommer att använda sig av. Man kan skriva felhanteringsrutiner till egendefinierade felkoder.

Om man får ett fel som det inte finns något felmeddelande till så skrivs "Unprintable error" ut på skärmen.

ERROR-satser som det inte finns någon felhanteringsrutin till ger felmeddelande och avbryter programkörningen.

Exempel: (i interaktivt läge)

```
ERROR 15
String too long
```

EXP(X)

Ger resultatet av e (naturliga logaritmens bas) upphöjt till X, dvs 2.7182818284590^X.

Beräkningen av EXP sker i antingen enkel eller dubbel precision beroende på om X är i enkel eller dubbel precision.

Om X är större än 88 för tal i enkel precision, eller större än 709 för tal i dubbel precision, så får man ett "Overflow" ("spill") felmeddelande och det största tal som maskinen kan framställa skrivs ut. Sedan fortsätter programkörningen.

Exempel:

```
FOR I=0 TO 3
   PRINT EXP(I)
NEXT I
```

Följande visas på skärmen:

```
1
2.718282
7.389056
20.08554
```

FIELD *[#]filnummer,filbredd* AS *strängvariabel...*

Reserverar utrymme för variabler i en direktåtkomstbuffert.

Det är god programmeringssed att ha FIELD-satsen så nära den sats som öppnade filen som möjligt.

filnummer motsvarar det nummer som angavs i OPEN-satsen.

filbredd är det antal tecken som skall reserveras för *strängvariabel.*

Man får inte reservera fler bytes (tecken) i FIELD-satsen än den postlängd som tidigare angivits i OPEN-satsen.

Man får använda hur många FIELD-satser som helst för samma fil. Alla utförda FIELD-satser kommer att ha effekt samtidigt.

Anmärkning: Använd inte samma variabler i INPUT- och LET-satserna som i FIELD-satser. När man använt FIELD med en variabel så pekar variabelns pekare på rätt plats i direktåtkomstbufferten. Om variabeln sedan används i en INPUT- eller LET-sats så kommer dess pekare inte längre att avse en plats i direktåtkomstbufferten.

Se även: GET, LSET, OPEN, PUT, RSET

Exempel:

```
OPEN "Betalningar" AS #1
```

FILES *[sträng]*

Visar alla filnamn i en given katalog.

Om *sträng* utelämnas så visas filnamnen i den aktuella katalogen. Om *sträng* är ett katalognamn så visas katalogens filnamn. Om *sträng* är ett filnamn så visas filnamnet om det existerar.

Exempel:

```
FILES "df1:"
FILES "c"
```

FIX(X)

Returnerar X:s heltalsdel som framställs genom trunkering av X.

FIX(X) är detsamma som SGN(X)*INT(ABS(X)). Skillnaden mellan FIX och INT är att FIX inte rundar av negativa tal till det närmaste lägre talet (se exemplet nedan).

Se även: CINT, INT

Exempel:

```
30 PRINT FIX(-58.75)
40 PRINT INT(-58.75)
```

Följande visas på skärmen:

```
-58
-59
```

FOR *variabel=x* **TO** *y* [STEP z]
NEXT *[variabel][,variabel...]*

Utför en slinga ett givet antal gånger.

FOR använder *x, y,* och *z* som numeriska uttryck och *variabel* som räknare. x är räknarens startvärde och y dess slutvärde.

Efter det att satserna mellan FOR och NEXT utförts så kommer räknaren att räknas upp (ned) med det värde som angivits efter STEP. Räknarens värde testas också så att att den inte överskrider sitt slutvärde y. Om räknaren fortfarande är mindre än y så sker ett återhopp till satsen efter FOR. Man brukar kalla detta en FOR...NEXT-slinga.

Om STEP inte är specificerat så sätts STEP till 1. Om STEP är negativ så kommer räknaren att räknas ned med motsvarande värde. Slingan kommer då att genomföras tills räknaren är mindre än y.

En FOR-sats utan NEXT-sats ger ett "FOR without NEXT" (FOR utan NEXT) felmeddelande. En NEXT-sats utan FOR ger ett "NEXT without FOR" felmeddelande.

Nästlade slingor

FOR...NEXT-slingor kan nästlas, dvs en FOR...NEXT-slinga kan placeras inuti en annan FOR...NEXT-slinga. När slingor nästlas måste varje slinga ha en unikt variabelnamn på sin räknare. NEXT-satsen för den inre slingan måste komma före den yttre slingans NEXT.

Man kan utelämna variablerna i NEXT-satsen. I sådant fall kommer NEXT att hänföras till den närmaste FOR-satsen.

Exempel:

```
FOR I=0 TO 100 STEP 10
PRINT I;
NEXT I
```

Följande visas på skärmen:

```
0 10 20 30 40 50 60 70 80 90 100
```

FRE *{-1}{-2}{x}*

Ger antalet fria bytes (tecken) i en angiven minnesarea.

FRE(-1) ger det totala antalet fria bytes i systemet.

FRE(-2) ger det antal bytes som stacken **inte** använt sig av.

FRE(x) (x skilt i från -1 och -2) ger antalet fria bytes i Amiga Basic minnet.

Se även: CLEAR

Exempel:

```
DEF FNMinnekvar&=FRE(0)-INT(Bobhöger+16)/16)*2*(BobBottom+1)*5-6
```

GET *[#]filnummer[,postnummer]* (1)
GET *(x1,y1)-(x2,y2),variabelnamn [(index[,index...,index])]* (2)

De två syntaxerna ovan hänför sig två olika användningsområden av GET:

1. Fil-GET: Läser en post från en direktåtkomstfil till direktåtkomstbufferten.

2. Skärm-GET: Läser av ett område av skärmen.

Fil-GET

filnummer anger det filnummer som filen öppnades under när den skapades med OPEN.

postnummer anger vilken post som skall läsas in. Utelämnas *postnummer* läses nästa post in. Det största postnumret man kan använda sig av är 16 777 215.

Då en GET-sats utförts kan informationen i posten nås genom att man använder variabler som preparerats med FIELD. Se "Direktåtkomstfiler" i kapitel 5, för närmare beskrivning av direktåtkomstfiler. Man kan också använda INPUT# och LINE INPUT# för att komma åt data i direktåtkomstbufferten.

Använd EOF(*filnummer*) för att kontrollera om GET försöker läsa förbi filslutet.

Skärm-GET

Skärm-GET används för att läsa av bilder som finns på skärmen. GET kan läsa av bitarna hos ett skärmområde. GET:s motsats är PUT som tilldelar bitarna i ett skärmområde nya värden.

I GET specificerar man ett rektangulärt område på skärmen med (x1,y1)-(x2,y2). De två punkterna är rektangelns vänstra övre hörn respektive högra nedre hörn.

variabelnamn är namnet på den tvådimensionella variabel som skall lagra det avlästa området. Variabeln kan vara vilken typ som helst förutom sträng. Indexet måste vara så stort att den indexerade variabeln

kan lagra hela bilden.

Genom flera *index* kan man lagra flera bilder i en flerdimensionell variabel. På så sätt kan man snabbt hoppa mellan olika bilder.

Om man använder en variabel av typen heltal så kan man analysera variabelns innehåll. Andra typer ger ett innehåll som inte går att analysera. Se nedan.

Variabelns storlek, i bytes, måste vara:

```
6+((y2-y1+1)*2*INT((x2-x1+16)/16))*D
```

där x och y är längden på rektangelns vågräta respektive lodräta sidor. D är skärmdjupet (förvalt värde 2).

Antalet bytes per element i en indexerad variabel är:

* 2 bytes för heltal.

* 4 bytes för enkel precision.

* 8 bytes för dubbel precision.

Exempel:

Satsen:

```
GET (10,20)-(30,40),MATRIS%
```

kräver 6+(40-20+1)*2*(INT(30-10)+16)/16))*2 bytes, dvs 174 bytes. Man behöver alltså en heltalsvariabel med minst 87 element.

Man kan studera x och y och även själva datat om man lagrar resultatet från GET i en heltalsvariabel. Höjden, längden och djupet hos rektangeln finns lagrat i elementen 0, 1, och 2.

Man använder GET och PUT tillsammans för att skicka bilder till och från skärmen. GET lagrar ett specificerat område av skärmen till en variabel. PUT skickar en bild som är lagrad i en variabel till skärmen.

Se även: PUT

Exempel:

```
GET (0,0)-(127,127),P
```

GOSUB *rad*
:

:
RETURN *[rad]*

Anropar en subrutin respektive hoppar tillbaka från en subrutin.

rad är radnumret eller radadressen för subrutinens första rad.

Återhopp sker när en RETURN-instruktion påträffas i subrutinen. Återhopp sker till satsen just efter den anropande GOSUB-satsen.

Man kan anropa en subrutin hur många gånger som helst. Man kan nästla subrutiner, dvs man kan ha subrutinanrop inuti en subrutin. Enda begränsningen är minnesstorleken. En subrutin får innehålla flera RETURN-satser.

rad gör att återhopp sker till den angivna raden eller radadressen. Använd dock denna typ av återhopp med försiktighet då andra GOSUB-satser, WHILE- och FOR-satser som var aktiverade när GOSUB utfördes kommer att fortsätta att vara aktiverade. Man kan då få felmeddelande av typen "FOR without NEXT".

Subrutiner kan skrivas in var som helst i programtexten, men det är god programmeringsvana att placera dem väl åtskilda från resten av programmet. För att undvika att de körs av misstag bör de föregås av en STOP- eller END-instruktion, eller en GOTO-sats som styr programflödet förbi subrutinen.

Exempel:

```
GOSUB InitGraphics
     .
     .
     .
InitGraphics:
  iDraw=30
  iErase=0
RETURN
```

GOTO *rad*

Ger hopp till angiven rad.

Det kan rekommenderas att man använder de strukturerade instruktionerna IF...THEN...ELSE, WHILE...WEND och ON...GOTO istället för GOTO för att styra programflödet. GOTO ger svårlästa och svårrättade program.

Exempel:

```
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

HEX$(X)

Ger en sträng om innehåller det hexadecimala värdet av det decimala uttrycket X. X avrundas till ett heltal innan HEX$ utförs.

Exempel:

```
FOR a=10 TO 16
PRINT a ; HEX$(a)
NEXT a
```

Följande visas på skärmen:

```
10 A
11 B
12 C
13 D
14 E
15 F
16 10
```

IF *uttryck* **{THEN** *sats1***}{GOTO** *rad***}**
[ELSE *sats2***]** (1)

IF *uttryck* **THEN** (2)
block1
[ELSEIF *uttryck* **THEN**
*block2***]**
[ELSE
*block3***]**
ENDIF

Styr programflödet beroende på resultatet av *uttryck.*

Följande regler gäller syntax 1:

* Om *uttryck* är sant så utförs GOTO-satsen eller *sats1*.
* Om uttrycket är falskt så utförs *sats2* om den finns annars går programkörningen vidare med satsen på nästa rad.
* *sats1* och *sats2* kan innehålla flera instruktioner och funktioner så länge hela satsen inte blir längre än radens maximala längd.
* THEN följs av en sats, ett radnummer eller en radadress.
* GOTO följs alltid av ett radnummer eller en radadress.
* Om antalet THEN och ELSE i satsen inte är lika så kommer ELSE att hänga ihop med den närmaste THEN-satsen som inte hänger ihop med någon annan ELSE.
* Om man i interaktivt läge anger ett radnummer eller en radadress i en IF...THEN...ELSE-sats och radnumret/radadressen inte finns i programmet så får man felmeddelandet "Undefined line number" ("Odefinierat radnummer").

Reglerna som gäller för syntax 1 gäller även för syntax 2 med följande tillägg:

* *block* kan innehålla nästlade IF...THEN...ELSE-satser. *block* får innehålla flera satser på en eller flera rader.

* Om *uttryck* är sant utförs *block1*. Därefter fortsätter programkörningen med första satsen efter END IF-satsen.

* Om inget uttryck är sant så kommer programkörningen att fortsätta med satsen efter END IF eller så utförs *block3* om det existerar.

* Man kan ha hur många ELSEIF-block som helst. ELSEIF är valfritt så man kan utesluta det helt om man vill.

* ELSE är tillval, d v s det är inte nödvändigt att ha det alternativet.

* Om THEN följs av någon sats tolkas den som ett enkelt IF...THEN...ELSE (syntax 1). Undantag är om THEN följs av en kommentar (se REM).

* Ett block-IF behöver inte vara den första satsen på raden.

Exempel:

```
INPUT a,b
IF a = 1 THEN
   IF b = 1 THEN
      PRINT "a and b are 1"
   ELSE
      PRINT "a = 1,b <> 1"
   END IF
ELSEIF a > 0 THEN
   IF b > 0 THEN PRINT "both a and b > 0"
   REM---above line is single-line-IF, not Block-IF
   PRINT "a > 0"
ELSE
   PRINT "a <= 0"
   PRINT "we know nothing about b"
END IF
```

INKEY$

Ger antingen en sträng bestående av ett tecken som läses från tangentbordet eller en tom sträng om ingen tangent är nedtryckt.

Tecknet skickas inte till bildskärmen. Alla tecken utom Amiga-punkt och CTRL-C (vilka avbryter programmet) kan hanteras.

Observera att om inget resultatfönster är aktivt så kommer alla tangentnedtryckningar att ignoreras. Tangentnedtryckningar kan bara skickas till ett aktivt resultatfönster, så det enda som händer är att skärmen blinkar till en gång.

Se även: SLEEP

Exempel:

```
GetAKey:
    PRINT "Current file is not saved."
    PRINT "Do you want to save it?"
    PRINT " Press Y key if you want to save it"
    PRINT " Press N key if you don't want to save it"
    PRINT " Press C key if you want to cancel command"
    Response=0
    WHILE Response=0
      a$=INKEY$
      IF a$<>"" THEN
        a$=UCASE$(a$)
        IF a$="Y" THEN Response=1
        IF a$="N" THEN Response=2
        IF a$="C" THEN Response=3
        IF Response=0 THEN BEEP
        PRINT Response
      END IF
    WEND
```

INPUT [;]*[ledtext;]variabellista*

Används för inmatning av data från tangentbordet under programkörning.

När en INPUT-sats utförs så stannar programkörningen och ett frågetecken skrivs ut på skärmen för att ange att programmet väntar på att man skall mata in data. Om en *ledtext* är angiven så kommer den att skrivas ut före frågetecknet.

Man kan använda kommatecken istället för semikolon. I så fall undertrycks frågetecknet. Till exempel så skulle satsen:

```
INPUT "PERSONNUMMER",B$
```

skriva ut ledtexten "PERSONNUMMER" utan ett efterföljande frågetecken.

Det som matas in lagras i de variabler som finns angivna i *variabellista*. Antalet element som matas in måste vara detsamma som antalet variabler i listan. Posterna separeras från varandra med kommatecken.

Variablerna i *variabellista* kan vara numeriska- eller strängvariabler. Indexering är tillåten. De inmatade elementens typer måste överensstämma med variablernas typ.

När man matar in strängar till INPUT-satser behöver de inte vara omgivna av citationstecken.

Om man matar in ett felaktigt antal element eller element av felaktig typ så kommer texten "?Redo from start" att skrivas ut på skärmen. INPUT-satsen körs från början. Ingen lagring av inmatade värden sker förrän en korrekt inmatning skett.

Exempel:

```
' THIS PROGRAM CONVERTS DECIMAL VALUES TO HEXADECIMAL
ANSWER$="Y"
WHILE (ANSWER$="Y")
  INPUT "ENTER DECIMAL NUMBER ", DECIMAL
  PRINT "HEX VALUE OF " DECIMAL "IS " HEX$(DECIMAL)
  PRINT "OCTAL VALUE OF "  DECIMAL "IS " OCT$(DECIMAL)
  INPUT "DO YOU WANT TO CONVERT ANOTHER NUMBER?  ", ANSWER$
  ANSWER$ = UCASE$(ANSWER$)
WEND
END
```

Vid en programkörning skulle man kunna få följande resultat:

```
ENTER DECIMAL NUMBER 16
HEX VALUE OF 16 IS 10
OCTAL VALUE OF 16 IS 20
DO YOU WANT TO CONVERT ANOTHER NUMBER? Y
ENTER DECIMAL NUMBER 31
HEX VALUE OF 31 IS 1F
OCTAL VALUE OF 31 IS 37
DO YOU WANT TO CONVERT ANOTHER NUMBER? N
```

INPUT$(X[,*[#]filnummer*]

Ger en sträng innehållande X tecken lästa från *filnummer*. Om *filnummer* inte är angiven så läses tecknen från tangentbordet.

När data läses från tangentbordet skickas inga tecken till bildskärmen. Alla kontrolltecken kan hanteras förutom CTRL-C som avbryter körningen av en INPUT$-sats.

Exempel:

```
objAttributes$=INPUT$(LOF(1),1)
OBJECT.SHAPE 1,objAttributes$
```

INPUT *#filnummer,variabellista*

Läser poster från en sekventiell fil och lagrar dem i variabler.

filnummer motsvarar det nummer som angavs när filen skapades med "OPEN".

variabellistan anger de variabler som elementen skall lagras i; variabeltyperna måste överensstämma med elementens datatyper.

Elementen skall ha samma format som om de matades in som svar på en INPUT-sats. Inledande blanktecken, radframatnings- och <RETUR>-tecken ignoreras. Vid numeriska element behandlas alla andra tecken som den första siffran i ett tal. Nästa blank-, <RETUR>-, radframmatnings- eller kommatecken skiljer sista siffran i talet från första siffran i ett nytt tal.

Om strängar inleds med citationstecken, ("), kommer nästa citationstecken att avsluta inläsning av strängen. Strängar som skall läsas in kan alltså inte innehålla citationstecken.

Om det första tecknet inte är ett citationstecken så kommer ett komma-, <RETUR>- eller radframmatningstecken att avsluta inläsning av elementet. Om elementets längd blir 255 tecken så kommer det 255:e tecknet att avsluta elementet.

INSTR([I,]X$,Y$)

Ger positionen för första förekomsten av Y$ i X$. I anger var i X$ som sökningen skall börja. I är tillval.

Om I är större än antalet tecken i X$, eller om X$ är en tom sträng, eller om Y$ inte finns i X$, så ger INSTR värdet 0. Om Y$ är en tom sträng så ger INSTR värdet I eller 1. X$ och Y$ kan vara strängvariabler, stränguttryck eller en sträng av tecken.

Exempel:

```
'THIS ROUTINE CHANGES THE ADDRESS FIELD IN RECORD$
RECORD$ ="n:JOHN JONES adr:3633 6TH ST WACO, TX           "
PRINT "RECORD$ =          "RECORD$
OFFSET = INSTR(RECORD$,"adr:") 'FIND START OF ADDRESS adr:
MID$(RECORD$,OFFSET,40) = "adr:222 ELM ST. WAXAHACHIE, TX         "
PRINT "MODIFIED RECORD$ = "RECORD$
```

Följande visas på skärmen:

```
RECORD$=          n:JOHN JONES adr: 3633 6TH ST WACO, TX
MODIFIED RECORD$= n:JOHN JONES adr: 222 ELM ST. WAXACHIE, TX
```

INT(X)

Ger det största heltal som är mindre än eller lika med X.

Se även: CINT, FIX

Exempel:

```
PRINT INT(3.4)
X=INT(37.98)
PRINT INT(X)
Y=INT(-32.3)
PRINT INT(Y)
```

Följande heltal skrivs ut på skärmen:

```
3
37
-33
```

KILL *filspec*

Raderar en fil från skiva. Om man försöker radera en fil som är öppen får man ett "File already open" ("Filen redan öppnad") felmeddelande.

filspec är ett korrekt filnamn omgivet av citationstecken.

Exempel:

```
KILL "MailLabels"
```

{LBOUND}{UBOUND}*(variabelnamn[,dimension])*

Ger undre eller övre gräns för indexet hos en indexerad variabel.

variabelnamn är den indexerade variabelns namn.

dimension använder man när variabeln är flerdimensionell. *dimension* anger då vilken av dimensionernas indexgränser som skall testas. Förvalt värde är 1.

Indexets undre gräns beror på om OPTION.BASE är satt till 0 eller 1.

Exempel:

LBOUND och UBOUND är speciellt användbara för att bestämma storleken av en indexerad variabel som skickats till ett underprogram. Istället för att skicka värdet på övre indexet kan man använda UBOUND i rutinen:

```
CALL INCREMENT (ARRAY1(0), ARRAY2(), TOTAL())

SUB INCREMENT (A(2), B(2), C(2)) STATIC
   FOR I = LBOUND(A,1) TO UBOUND (A,1)
   FOR J = LBOUND(A,2) TO UBOUND(A,2)
     C(I,J) = A(I,J) + B(I,J)
   NEXT J
  NEXT I
END SUB
```

**LEFT$**(X$,I)

Ger en sträng som består av I tecken från vänster i X$. Om I är större än antalet tecken i X$ så returneras hela X$. Om I är 0 så returneras en tom sträng.

Se även: MID$, RIGHT$

LEN(X$)

Ger antalet tecken i X$. Alla typer av tecken räknas, även blanka och sådana tecken som inte skrivs ut.

Exempel:

```
'THIS ROUTINE EXRACTS THE ADDRESS a: FROM STRING RECORD$
'
RECORD$ = "n:JOHN JONES ss:5349 12 99 a:3633 6TH ST WACO,TX"
LENGTH = LEN(RECORD$)  'DETERMINE LENGTH OF RECORD
OFFSET = INSTR(RECORD$,"a:") 'FIND START OF ADDRESS a:
RIGHTCHAR = LENGTH - OFFSET - 1
ADDRESS$ = RIGHT$(RECORD$,RIGHTCHAR) 'EXTRACT ADDRESS
FROM RECORD$
PRINT ADDRESS$
```

Följande visas på skärmen:

```
3633 6TH ST WACO, TX
```

[LET] *variabelnamn=uttryck*

Lagrar värdet av ett uttryck i en variabel.

Observera att ordet LET inte behöver tas med i instruktionen. Lika-med-tecknet räcker för att tilldela en variabel ett värde.

Exempel:

Följande exempel belyser att LET är tillval i instruktionen:

```
10 LET a=1 : LET b=1 : LET c=3
20     d=1 :     e=2 :     f=3
30 PRINT a b c d e f
```

Utskriften blir:

```
1 2 3 1 2 3
```

LIBRARY *{"filnamn"}*{CLOSE}

LIBRARY *filnamn* öppnar ett bibliotek med maskinspråksrutiner (underprogram och funktioner) för användning från Amiga Basic.

LIBRARY CLOSE stänger alla öppnade bibliotek.

filnamn är ett stränguttryck som anger i vilken fil som Amiga Basic skall leta efter maskinspråksrutinerna. Man kan öppna upp till fem bibliotek samtidigt. Dessa bibliotek kommer att vara öppna ända tills en NEW, RUN, eller LIBRARY CLOSE-sats utförs. Se appendix F för mer information om dessa instruktioner.

Man kan få två felmeddelanden vid användning av LIBRARY instruktionen: "File not found" ("Hittar ej filen") och "Out of memory" ("Slut på minnet").

För att kunna använda LIBRARY-instruktionen måste man skapa en speciell fil med tillägget ".bmap". Denna fil beskriver rutinerna i det angivna biblioteket. Se appendix F för förklaring om hur man skapar en ".bmap"-fil.

Exempel:

```
LIBRARY "graphics.library"
CALL SetDrMd& (WINDOW(8),3)
```

LINE [[STEP](x1,y1)]-[STEP]*(x2,y2),[färgid][,b[f]]*

Ritar en linje eller rektangel i det aktuella resultatfönstret.

Koordinaten för linjens startpunkt är x1,y1; koordinaten för linjens slutpunkt är x2,y2.

färgid anger vilken färg som skall användas; den refererar till *färgid* i en PALETTE-sats.

Om "b"-tillvalet anges så ritas en rektangel upp med x1,y1 och x2,y2 som motstående hörn.

Anges "bf" så kommer insidan av rektangeln att fyllas med förgrundens färg.

Om man anger koordinater som inte ryms i resultatfönstret så kommer den koordinat som är för stor att minskas till det närmast giltiga värdet.

Med STEP kan man ange relativa koordinater. Om senast angivna punkt är 10,10 så kommer satsen LINE STEP (10,5) att referera till punkten 20,15.

Om STEP används bara vid den andra koordinaten i LINE-instruktionen så kommer den koordinaten att vara relativ den första koordinaten.

Exempel:

```
LINE(0,0)-(120,120),,BF
```

Satsen fyller en rektangel med förgrundens färg som tidigare specificerats med en "COLOR"-sats eller av Amiga systemet inställning.

LINE INPUT [;]*[ledtext;]strängvariabel*

Läser en hel rad från tangentbordet och placerar den utan citationstecken i en strängvariabel.

ledtext skrivs ut då inmatning kan ske. Frågetecken skrivs bara ut om de ingår i *ledtext*.

Alla data som matas in före <RETUR> kommer att lagras.

Om LINE INPUT följs av ett semikolon så kommer inte <RETUR> att resultera i någon radframmatning på skärmen.

Man kan avbryta en LINE INPUT-sats genom att trycka Amiga-punkt.

Exempel:

```
OPEN "O",#2,"INFO"
    LINE INPUT "Customer Data?";CUSTOMER$
    PRINT #2, CUSTOMER$
CLOSE #2
OPEN "I",#2,"INFO"
     LINE INPUT #2,CLIENT$
PRINT CLIENT$
END
```

Följande visas på skärmen:

```
Customer Data? Clarissa Dalloway $10.17 Penknife
Clarissa Dalloway $10.17 Penknife
```

LINE INPUT# *filnummer;strängvariabel*

Läser en hel rad från en sekventiell fil och lagrar den utan citationstecken i en strängvariabel.

filnummer motsvarar det filnummer som användes i OPEN-satsen när filen skapades.

strängvariabel är namnet på den variabel som den inlästa raden kommer att lagras i.

Ett <RETUR>-tecken avgränsar varje rad i filen. LINE INPUT# läser bara in de tecken som kommer före <RETUR>-tecknet och hoppar sedan över detta tecken och det efterföljande radframmatningstecknet innan den läser nästa rad.

LINE INPUT# är användbart om varje rad i en fil är uppdelad i fält eller om ett program som är sparat som en ASCII-fil skall läsas av ett annat program.

Se även: LINE INPUT, SAVE

Exempel:

Se exempel under LINE INPUT

LIST *[rad][[-[rad]],"filnamn"*

Listar programmet som finns i minnet i List-fönstret, till en fil eller till någon annan ansluten utrustning.

rad är ett radnummer eller en radadress.

Om bara första *rad* anges kommer den raden och alla efterföljande rader att visas.

Om bara andra *rad* anges kommer alla rader från början av programmet och fram till och med den raden att visas.

Anges båda *rad* så kommer alla rader mellan dessa rader att visas.

Om *filnamn* är ett stränguttryck av typen "SCRN:" eller "LPT1" så kommer listningen att ske till angiven utrustning.

Se även: "Redigera program" i kapitel 4.

Exempel:

```
LIST
```

LLIST *[rad][-[rad]]*

Listar programmet i minnet på skrivaren (PRT:).

Tillvalen till LLIST är detsamma som för LIST med det undantaget att man inte kan välja vilken utrustning som listningen skall ske på; den sker alltid på skrivaren.

Se även: LIST

LOAD *[filspec*[,R]*]*

Läser in ett program från skivan till minnet. Se SAVE för beskrivning av olika filspecifikationer (skivstation, katalog).

Om *filspec* inte anges så öppnas ett frågefönster som ber användaren ange ett namn på filen som skall läsas in.

R-tillvalet gör att programmet automatiskt körs då det läses in.

LOAD stänger alla öppna filer och raderar alla programrader och variabler i minnet. När man använder R-tillvalet så stängs dock inga öppna filer. Man kan alltså skicka information mellan olika program genom att använda deras datafiler.

Se även: CHAIN, MERGE, SAVE

LOC*(filnummer)*

När man arbetar med direktåtkomstfiler ger LOC postnumret på den senast lästa eller skrivna posten.

Vid sekventiella filer ger LOC antalet lästa eller skrivna poster. LOC räknas då ut genom att man tar antalet tecken skrivna eller lästa från filen dividerat med antingen den förvalda poststorleken (128 tecken) eller med den poststorlek man angivit i OPEN-satsen när man skapade filen:

LOC = Antal bitgrupper skrivna eller lästa \ Poststorleken

Observera att det är en heltalsdivision som utförs.

För filer öppnade till KYBD: eller COM1: ger LOC värdet 1 om något tecken är färdigt att läsas från filen, annars ges värdet 0.

När en fil öppnas för sekventiell läsning så läses automatiskt den första posten vilket gör att LOC sätts till 1 direkt. LOC förutsätter att *filnummer* är det nummer som filen öppnades med.

LOCATE *[rad][,kolumn]*

Flyttar pennan till angiven rad och kolumn i det aktuella resultatfönstret.

rad och *kolumn* måste vara större än eller lika med 1. Punkten som anges är relativ till det övre vänstra hörnet i det aktuella fönstret. Om

man inte anger några parametrar så används pennans aktuella position.

För att beräkna *rad* och *kolumn* använder LOCATE höjden och bredden på tecknet "0" (noll) vilka beror på vilken teckenuppsättning som används i det aktuella resultatfönstret.

Exempel:

```
Y = CSRLIN   ' GET CURRENT CURSOR LINE NUMBER (VERTICAL POSITION)
X = POS(0)   ' GET CURRENT CURSOR COLUMN NUMBER (HORIZONTAL POSITION)
LOCATE 20,1 ' PLACE CURSOR ON LINE 24, ROW 1 (BOTTOM OF SCREEN)
PRINT "THIS PRINTS AT LOCATION 20,1 (BOTTOM OF PAGE)"
LOCATE Y,X   ' PLACE CURSOR IN ORIGINAL LOCATION
PRINT "THIS PRINTS AT ORIGINAL LOCATION OF CURSOR"
```

LOF*(filnummer)*

Ger längden på en fil i antalet tecken.

Vid filer som är öppnade till SCRN:, KYBD: eller LPT1: ger LOF alltid värdet 0.

Exempel:

```
entireFile$=INPUT$(LOF(1),1)
```

LOG(X)

Ger den naturliga (basen e) logaritmen av X. X måste vara större än 0.

Beräkningen sker i enkel eller dubbel precision beroende på om X är i enkel eller dubbel precision.

Exempel:

```
10 FOR I=1 TO 2 STEP .2
20 PRINT "LOG OF "I "= ";LOG(I)
30 NEXT I
40 END
```

Följande visas på skärmen:

```
LOG OF  1 =  0
LOG OF  1.2 = .1823216
LOG OF  1.4 = .3364723
LOG OF  1.6 = .4700037
LOG OF  1.8 = .5877868
```

LPOS(X)

Ger skrivar-huvudets aktuella (skenbara) position i skrivar-bufferten.

LPOS ger inte alltid printerhuvudets fysiska position.

Exempel:

```
IF LPOS(x)>60 THEN PRINT CHR$(13)
```

LPRINT [[USING *stränguttryck;*]*uttryckslista*]

Skickar data till skrivaren.

LPRINT och LPRINT USING är detsamma som PRINT och PRINT USING förutom det att alla data skickas till skrivaren istället för till skärmen.

Exempel:

Se exemplen för PRINT och PRINT USING

LSET *strängvariabel=stränguttryck*

Flyttar data från minnet till en direktåtkomstbuffert för att förbereda för en PUT-sats.

Om *stränguttryck* innehåller färre tecken än vad som angivits i en tidigare FIELD-sats så vänsterjusterar LSET-strängen med blanktecken för att fylla ut det extra utrymmet. Om strängen är för lång så kommer strängen att huggas av så att tecken till höger förloras.

Numeriska värden måste konverteras till strängar innan de kan användas med LSET, se MKI$, MKL$, MKS$ och MKD$.

Anmärkning: Man kan använda LSET för att i ett givet fält vänsterjustera en sträng som inte är preparerad med en FIELD-instruktion.

Se även: RSET, MKI$, MKL$, MKS$ och MKD$.

MENU *{menyid, postid, status, [,titelsträng]}*{RESET}

Instruktionen används för att skapa menyer till menyraden eller att för återställa menyradens förvalda menyuppsättning.

menyid anger menyn. Det kan vara ett värde mellan 1 och 10.

postid anger de olika valen som finns på menyn. Det kan vara ett värde mellan 0 och 19. Om *postid* är mellan 1 och 19 avses något av valen på menyn, är *postid* 0 avses hela menyn.

Om *status* är 0 så kopplas menyn eller en menypost bort, 1 gör den tillgänglig igen, 2 gör den tillgänglig och placerar ett kontrollmärke bredvid den. Om *postid* är 0 så kommer *status* att påverka hela menyn. Om man har för avsikt att placera ett kontrollmärke före en menypost så måste man lämna ett utrymme på två mellanslag framför menyposten. Där kommer kontrollmärket att placeras.

titelsträng är namnet på menyraden eller på en menypost.

MENU RESET återställer menyraden till Amiga Basic:s förvalda menyrad.

Se även: MENU(), MENU ON, ON MENU, SLEEP

Exempel:

```
MENU 1,0,1,"Transactions:"
MENU 1,1,1,"Deposits"
MENU 1,2,1,"Withdrawals"
MENU 1,3,1,"Automatic Payment"
MENU 1,5,1,"Credit Card Purchase"
```

MENU({0}{1})

MENU(0) ger numret på den senast använda menyn. MENU(0) nollställs alltid då det utförts så att man kan avläsa menyraden på samma sätt som INKEY$ avläser tangentbordet.

MENU(1) ger numret på den senast valda menyposten.

Se även: MENU, MENU ON, ON MENU, SLEEP

MENU {ON}{OFF}{STOP}

Sätter på (ON), stänger av (OFF) och avbryter tillfälligt (STOP) avbrottshantering av menyhändelser. En menyhändelse inträffar då användaren väljer en meny definierad av en MENU-sats. MENU()-funktionen kan användas för bestämma vilken meny eller menypost som har valts.

Vid MENU STOP kommer ingen ON MENU...GOSUB att utföras. Dock kommer användarens val av menyer att lagras så att så fort en MENU ON-sats utförs så kommer en ON MENU...GOSUB-sats att utföras om användaren valt någon meny under den tid som avbrottshanteringen tillfälligt var avbruten.

Se även: MENU, MENU(), ON MENU, "Avbrottshantering" i kapitel 6.

Exempel:

```
ON MENU GOSUB CheckMenu
ON MOUSE GOSUB CheckMouse
MENU ON
MOUSE ON
```

MERGE *filspec*

Lägger till en fil (som finns på skiva) till programmet i minnet. Filen som skall läggas till måste vara sparad som en ASCII-fil, annars får man felmeddelandet "Bad file mode" ("Felaktigt filformat"). Se SAVE för hur man sparar en fil i ASCII-format. Då satsen är utförd återvänder Amiga Basic till kommandoläget.

Exempel:

```
MERGE "SortRoutine"
```

MID$*(stränguttryck1,n [,m])=stränguttryck2*

Satsen ersätter en bit av en sträng med en annan sträng.

n och *m* är heltalsuttryck, *stränguttryck1* och *stränguttryck2* är stränguttryck. Tecknen i *stränguttryck1*, med början i position *n*, ersätts med tecknen i *stränguttryck2*.

Tillvalet *m* anger hur många av tecknen i *stränguttryck2* som skall användas. Om *m* inte anges så används alla tecken i *stränguttryck2*. Man kan inte ersätta tecken i *stränguttryck1* så att dess ursprungliga längd överskrids.

Exempel:

```
'THIS ROUTINE CHANGES THE ADDRESS FIELD IN RECORD$
'
RECORD$ ="n:JOHN JONES adr:3633 6TH ST WACO, TX       "
PRINT "RECORD$ =          ",RECORD$
OFFSET = INSTR(RECORD$,"adr:") 'FIND START OF ADDRESS adr:
MID$(RECORD$,OFFSET,40) = "adr:222 ELM ST. WAXAHACHIE, TX "
PRINT "MODIFIED RECORD$ = ",RECORD$
```

Följande visas på skärmen:

```
RECORD$ =           n:JOHN JONES adr:3633 6TH ST WACO, TX
MODIFIED RECORD$ =  n:JOHN JONES adr:222 ELM ST. WAXAHACHIE, TX
```

MID$(X$,*n [,m]*)

Ger en sträng med *m* tecken ur strängen X$ med början från tecken *n*.

Om *n* är större än X$ längd så returneras en tom sträng.

n och *m* måste vara ett värde mellan 1 och 32767. Om *m* inte anges eller om det är färre än *m* tecken till höger om tecken *n* så returneras alla tecken till höger med början från tecken *n*.

MKI$*(kort heltalsuttryck)*
MKL$*(långt heltalsuttryck)*
MKS$*(uttryck i enkel precision)*
MKD$*(uttryck i dubbel precision)*

Konverterar tal till strängar så att de kan skrivas till en direktåtkomstfil.

MKI$ konverterar ett kort heltal till en 2-bytessträng.

MKL$ konverterar ett långt heltal till en 4-bytessträng.

MKS$ konverterar ett tal i enkel precision till en 4-bytessträng.

MKD$ konverterar ett tal i dubbel precision till en 8-bytessträng.

Man måste konvertera numeriska värden till strängar innan man kan skriva dem till en direktåtkomstfil.

Istället för att som STR$ konvertera ett tals binära värde till dess motsvarighet i strängform, så flyttar MK$ det binära värdet in i en sträng med korrekt längd. Det sparar mycket minnesutrymme.

Se även: CVI, CVL, CVS, CVD, LSET, RSET, kapitel 5, "Att arbeta med filer och logiska enheter"

Exempel 1:

```
PRINT #1, MKI$(Flags);
```

Exempel 2:

```
OPEN "AccountInfo" AS #2 LEN = 14
   FIELD #2,8 AS ACCT$,4 AS CHECK$,2 AS DEPOSIT$
   GET #2,1
      LET ACCOUNTNO# = 987654332556#
      LET CHECKING! = 123456!
      LET SAVINGS% = 2500
      LSET ACCT$ =    MKD$(ACCOUNTNO#)
      LSET CHECK$ =   MKS$(CHECKING!)
      LSET DEPOSIT$ = MKI$(SAVINGS%)
   PUT #2,1
CLOSE #2
END
```

MOUSE*(n)*

MOUSE() ger information om musens valknapp (vänstra knappen) och om musmarkörens läge i det aktuella fönstret. MOUSE() ger ingen information om menyknappen (högra knappen), se istället MENU().

MOUSE() har sju olika funktioner. Genom att ange ett tal mellan 0 och 7 kommer man åt de olika funktionerna:

MOUSE(0): valknappens status

MOUSE(0) ger information om valknappens status. Efter ha utfört MOUSE(0) så behåller Amiga Basic musens start- och slutposition tills nästa MOUSE(0) utförs. Därför skall man, då man konstaterat musrörelse med MOUSE(0), använda MOUSE(3-6) för att bestämma rörelsens start- och slutposition.

MOUSE(0) ger följande värden:

Värde	Förklaring
0	Valknappen är inte nedtryckt och har inte varit nedtryckt sedan den senaste MOUSE(0)-satsen utfördes.
1	Valknappen är inte nedtryckt men har klickats en gång sedan den senaste MOUSE(0)-satsen. Använd MOUSE(3-5) för att bestämma valets start- och slutposition.
2	Valknappen är inte nedtryckt men har klickats två gånger sedan senaste MOUSE(0)-satsen. Använd MOUSE(3-5) för att bestämma valets start- och slutposition. På samma sätt anger en returnerad 3:a att valknappen klickats tre gånger.
-1	Valknappen är nedtryckt utan att ha blivit klickad tidigare. Oftast betyder detta att musen flyttas.
-2	Valknappen är nedtryckt efter att ha blivit klickad en gång tidigare. Oftast betyder detta att musen flyttas. På samma sätt anger -3 att valknappen klickats tre gånger.

MOUSE(1): aktuell x-koordinat

Ger den vågräta (x) koordinaten för musmarkörens position då senaste

MOUSE(0)-sats utfördes. Detta gäller oberoende av valknappens status.

MOUSE(2): aktuell y-koordinat

Ger den lodräta (y) koordinaten för musmarkörens position då senaste MOUSE(0)-sats utfördes. Detta gäller oberoende av valknappens status.

MOUSE(3): start x-koordinat

Ger den vågräta (x) koordinaten för musmarkörens position då valknappen klickades före senaste MOUSE(0)-sats.

MOUSE(4): start y-koordinat

Ger den lodräta (y) koordinaten för musmarkörens position då valknappen klickades före senaste MOUSE(0)-sats.

MOUSE(5): slut x-koordinat

Om valknappen var nedtryckt vid senaste MOUSE(0)-anrop så ges den vågräta (x) koordinaten för musmarkörens position vid MOUSE(0)-anropet. Om valknappen inte var nedtryckt så ges x-koordinaten för den position musmarkören var i då valknappen släpptes upp. Man använder MOUSE(5) för att ta reda på x-koordinaten för en musrörelses slutposition.

MOUSE(6): slut y-koordinat

Fungerar på samma sätt som MOUSE(5) förutom det att den ger den lodräta (y) koordinaten.

Exempel:

```
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

MOUSE {ON}{OFF}{STOP}

Sätter på (ON), stänger av (OFF), och avbryter tillfälligt (STOP) avbrottshantering baserad på klickning av musens valknapp.

Vid MOUSE STOP kommer ingen ON MOUSE...GOSUB-sats att utföras. Dock kommer användarens klickningar av valknappen att lagras så att så fort en MOUSE ON-sats utförs så kommer en ON MOUSE...GOSUB-sats att utföras om användaren klickat valknappen under den tid som avbrottshanteringen tillfälligt var avbruten.

Se även: MOUSE, ON MOUSE, "Avbrottshantering" i kapitel 6.

NAME *"nuvarande filnamn"* AS *"nytt filnamn"*

Ändrar namnet på en fil som finns lagrad på skiva.

Båda parametrarna är stränguttryck. *nuvarande filnamn* måste existera medan *nytt filnamn* inte får finnas sedan tidigare, i så fall får man ett felmeddelande.

Exempel:

```
NAME "Accounts" AS "LEDGER"
```

NEW

Tömmer minnet från program, raderar alla variabler, raderar List- och resultatfönstret. Används då man skall skriva ett nytt program.

NEW kan skrivas i resultatfönstret eller väljas från Project-menyn. Om programmet i minnet har ändrats sedan det lästes in så öppnas ett frågefönster som frågar om man vill spara den ändrade versionen. Om NEW utförs som en sats i ett program så kommer Amiga Basic att återvända till redigeringsläge.

NEW stänger alla filer och stänger av TRACE.

NEXT *[variabel][,variabel]]*

Anger slut på en slinga inledd av en FOR-sats.

Se även: FOR...NEXT

OBJECT.{AX}{AY} *objektid, värde*

Definierar ett objekts acceleration i x- eller y-led.

objektid refererar till *objektid* i en OBJECT.SHAPE-sats och anger för vilket objekt som accelerationen skall definieras.

värde anger accelerationen i bildelement per kvadratsekund.

OBJECT.CLIP *(x1,y1)-(x2,y2)*

Definierar en rektangel som anger var ett objekt får ritas.

x1,y1 anger rektangelns övre vänstra hörn, *x2,y2* anger det nedre högra hörnet.

OBJECT.CLIP:s förvalda värden är resultatfönstrets lister.

Anmärkning: Rektangelstorleken som definierats med OBJECT.CLIP förändras inte när man ändrar resultatfönstrets storlek med storleksrutan. Det betyder att om man förstorar resultatfönstret kommer objekten ändå att ritas upp inom den rektangel som angivits med OBJECT.CLIP.

OBJECT.CLOSE [objektid[,objektid...]]

Frigör det minne som används av ett eller flera objekt.

objektid refererar till *objektid* i en OPEN.SHAPE-sats i det den (eller de) identifierar ett eller flera objekt i det aktuella resultatfönstret.

Om *objektid* inte anges så frigörs minnet från alla objekt i det aktuella resultatfönstret.

OBJECT.HIT *objektid [objektfilter][,kollisionsobjektfilter]*

objektid refererar till *objektid* i en OBJECT.SHAPE- sats.

Om ingen OBJECT.HIT-sats utförs så kommer objekten att kollidera med varandra och med fönsterlisterna. Med OBJECT.HIT-satsen kan man få objekt att passera igenom varandra utan att kollidera.

objektfilter är ett 16-bitars filter som beskriver *objektid.* *kollisionsobjektfilter* beskriver det objektet som *objektid* skall kollidera med. Om man sätter den minst signifikanta biten i *kollisionsobjektfilter* till 1 så kommer *objektid* att kollidera med fönsterlisterna.

Om man utför ett logiskt "och" (AND) mellan ett objektfilter hos ett objekt med kollisionsobjektfiltret hos ett annat objekt, och resultatet är skilt från noll så betyder det att objektid kommer att kollidera med alla objekt som kollisionsobjektfilter anger. Detta resulterar i en COLLISION-händelse.

Mer information om objektfilter och kollisionsobjektfilter kan man hitta i handboken *Amiga ROM Kernel Manual.* De behandlas där under namnen "MeMask" och "Hitmask".

Exempel:

```
OBJECT.SHAPE 1,Asteroid$
OBJECT.SHAPE 2,Ship$
OBJECT.SHAPE 3,Missle$
OBJECT.HIT 1,8,7 'collides with border, ship, missle
OBJECT.HIT 2,2,9 'collides with border, asteroid
OBJECT.HIT 3,4,9 'collides with border, asteroid
```

OBJECT.ON *[objektid[,objektid...]]*
OBJECT.OFF *[objektid[,objektid...]]*

Gör ett objekt synligt (ON) eller osynligt (OFF).

objektid refererar till *objektid* i en OBJECT.SHAPE-sats; den anger vilket objekt i det aktuella resultatfönstret som skall göras synligt eller osynligt.

OBJECT.ON utan parametrar gör att alla objekt i det aktuella resultatfönstret blir synliga. Om objekten tidigare startats med OBJECT.START så kommer de åter att röra på sig.

OBJECT.OFF utan parametrar gör att alla objekt i det aktuella resultatfönstret blir osynliga. Objekt som är i rörelse stannas så att inga kollisioner uppstår.

Se även: OBJECT.START och OBJECT.STOP

Exempel:

Se OBJECT.SHAPE för exempel på OBJECT.ON

OBJECT.PLANES *objektid [,bitplan][,planpåav]*

Sätter en bobs *bitplan* och *planpåav* filter. För detaljer se *Amiga ROM Kernel Manual* där de behandlas under namnen PlanePick och PlaneOnOff.

bitplan anger vilket bitplan (skärmdjup) som kommer att påverkas när objektet ritas upp. *planpåav* anger vad som skall göras med de bitplan som inte påverkas av *bitplan*. *bitplan* och *planpåav* är heltal mellan 0 och 255.

OBJECT.PRIORITY *objektid, värde*

Anger prioriteten för när ett objekt skall ritas i förhållande till andra objekt (med lägre eller högre prioritet). Denna instruktion påverkar bara bobs, inte sprites.

Objekt med samma prioritet ritas i slumpmässig ordning.

objektid refererar till *objektid* i en OBJECT.SHAPE-sats; den anger det objekt som skall ritas.

värde är ett tal från -32768 till 32767 som anger prioriteten; ju högre tal desto högre prioritet. Ett objekt med prioriteten 8 visas framför ett objekt med prioriteten 7.

OBJECT.SHAPE

Syntax 1: OBJECT.SHAPE *objektid, definition*

OBJECT.SHAPE syntax 1 anger formen, färgen, positionen och en del egenskaper hos ett rörligt objekt. Detta gäller även blitter-objekt (bobs) och VSprites vilka diskuteras i kapitlet "Graphic Animation" i *Amiga ROM Kernel Manual.*

objektid anger objektet. OBJECT-satser refererar till objektet genom *objektid. objektid* är ett tal från 1 till n, där n bara begränsas av tillgängligt minnesutrymme.

definition är ett stränguttryck som beskriver objektets fasta attribut (t ex form, storlek och färg). Objekteditorn skapar detta stränguttryck, se kapitel 7.

Syntax 2: OBJECT.SHAPE *objektid1, objektid2*

OBJECT.SHAPE syntax 2 skapar ett nytt objekt genom att kopiera *objektid2*:s form till *objektid1*. Tvillingobjekten kommer att dela på samma minnesutrymme. Man spar sålunda minne genom att skapa flera objekt med denna sats.

Trots att *objektid2* och *objektid1* delar minne kan man tilldela dem olika attribut med hjälp av olika OBJECT-satser. Värdena hos OBJECT.X, OBJECT.Y, OBJECT.VX, OBJECT.VY, OBJECT.AX och OBJECT.AY sätts till 0 för att kunna användas till detta ändamål.

Exempel:

```
OPEN "ball" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
```

I ovanstående exempel är de fasta attributen lagrade i filen "ball" som tidigare skapats med Objekteditorn.

Det följande exemplet visar en rutin som startar och hanterar kollisioner för objektet som definierades i filen "ball".

Exempel:

```
        WINDOW 4, "Animation", (310, 95)-(580,170),15
        ON COLLISION GOSUB BounceOff
        COLLISION ON
        OPEN "ball" FOR INPUT AS 1 'file created by the Object Editor
        OBJECT.SHAPE 1,INPUT$(LOF(1),1)
        CLOSE 1
        OBJECT.X 1,10
        OBJECT.Y 1,50
        OBJECT.VX 1,30
        OBJECT.VY 1,30
        OBJECT ON
        OBJECT.START
    WHILE 1
        SLEEP
    WEND
    BounceOff:
        saveId=WINDOW(1)
        WINDOW 4
        i=COLLISION(0)
        IF i=0 THEN RETURN
        j=COLLISION(i)
        IF j=-1 OR j=-3 THEN
           'object bounced off left or right border
           OBJECT.VX i,-OBJECT.VX(i)
        ELSE
           'object bounced off top or bottom border
           OBJECT.VX i,-OBJECT.VX(i)
        END IF
        OBJECT.START
        WINDOW saveId
    RETURN
```

OBJECT.{START}{STOP} *[objektid [,objektid...]]*

Sätter ett eller flera objekt i rörelse (START), eller stannar ett eller flera objekt (STOP).

objektid refererar till *objektid* i en OPEN.SHAPE-sats; den (eller de) anger de objekt i det aktuella resultatfönstret som OBJECT.START eller OBJECT.STOP skall påverka.

OBJECT.START utan några parametrar sätter alla objekt i det aktuella resultatfönstret i rörelse.

OBJECT.STOP utan några parametrar stannar alla objekt i det aktuella resultatfönstret.

När två objekt kolliderar så utförs ett OBJECT.STOP på båda objekten.

När ett objekt kolliderar med en fönsterlist så utförs ett OBJECT.STOP på objektet.

Exempel:

Se OBJECT.SHAPE.

OBJECT.{VX}{VY} *objektid, värde*

Bestämmer hastigheten för ett objekt i x-led (VX) och y-led (VY).

objektid refererar till *objektid* i en OBJECT.SHAPE-sats; den identifierar objektet som skall få den angivna hastigheten.

värde anger hastigheten i bildelement per sekund.

Se även: OBJECT.AX, OBJECT.AY

Exempel:

```
OBJECT.VX 1,30
OBJECT.VY 1,30
```

Se även: OBJECT.SHAPE för exempel på hur man använder OBJECT.VX och OBJECT.VY med andra OBJECT-satser.

OBJECT.{VX}{VY}*(objektid)*

Ger hastigheten på ett objekt i x-led (VX) eller y-led (VY).

objektid refererar till *objektid* i en OBJECT.SHAPE-sats; den identifierar det objekt vars hastighet skall returneras.

Hastigheten ges i bildelement per sekund.

Se även: OBJECT.AX, OBJECT.AY

OBJECT.{X}{Y} *objektid, värde*

Anger punkten där ett objekt skall ritas och placerar objektet där. Punkten blir startpunkten för hanteringen av det rörliga objektet.

objektid refererar till *objektid* i en OPEN.SHAPE-sats; den identifierar det objekt vars rektangels övre vänstra hörn skall definieras.

värde anger x- eller y-koordinaten; den kan vara ett numeriskt uttryck mellan -32768 och 32767.

Man använder satsen för att ange en startpunkt för hantering av det rörliga objektet, eller för att under programkörningen flytta objektet; hanteringen av det rörliga objektet kommer då att fortsätta från den nya punkten.

Exempel:

```
OBJECT.X 1,10
OBJECT.Y 1,50
```

Se även OBJECT.SHAPE för exempel på hur man använder denna sats med andra OBJECT-satser.

OBJECT.{X}{Y}*(objektid)*

Ger x- eller y-koordinaten för det övre vänstra hörnet av ett objekts rektangel.

objektid refererar till *objektid* i en OPEN.SHAPE-sats. Koordinaterna för det övre vänstra hörnet av rektangeln till det angivna objektet returneras.

OCT$(X)

Returnerar en sträng som innehåller det oktala värdet av X. Före beräkningen sker avrundas X till ett heltal.

Se även: HEX$

Exempel:

```
' THIS PROGRAM CONVERTS DECIMAL VALUES TO HEXADECIMAL
ANSWER$="Y"
WHILE (ANSWER$="Y")
  INPUT "ENTER DECIMAL NUMBER ", DECIMAL
  PRINT "OCTAL VALUE OF "  DECIMAL "IS " OCT$(DECIMAL)
  INPUT "DO YOU WANT TO CONVERT ANOTHER NUMBER?  ", ANSWER$
WEND
END
```

Följande skulle kunna visas på skärmen när programmet körs:

```
ENTER DECIMAL NUMBER 16
OCTAL VALUE OF  16 IS 20
```

ON BREAK GOSUB *{adress}{0}*

Ger anrop till en viss rutin då användaren trycker Ctrl-C, Amiga-punkt eller väljer "Stop" från Run-menyn.

adress är en radadress eller ett radnummer i den subrutin till vilket ett hopp kommer att ske när användaren försöker stoppa programmet.

Se även: BREAK ON i kapitel 6.

Exempel:

```
  ON BREAK GOSUB 100
  BREAK ON
10 GOTO 10
    .
    .
    .
100 PRINT "Sorry, this program can't be stopped"
    RETURN
```

ON COLLISION GOSUB *{adress}{0}*

Ger anrop till en viss rutin när COLLISION-funktionen returnerar ett värde skilt från 0, dvs när ett objekt kolliderar med ett annat objekt eller med en av fönsterlisterna.

adress är en radadress eller ett radnummer i den subrutin till vilket ett eventuellt hopp sker. GOSUB 0 stänger av hanteringen av

kollisionshändelser, dvs ON COLLISION kommer inte ha någon funktion förrän en COLLISION ON-sats utförts.

Se även: "Avbrottshantering" i kapitel 6.

För exempel se OBJECT.SHAPE.

ON ERROR GOTO *rad*

Ger hopp till en felhanteringsrutin.

När felhantering är påsatt kommer alla fel att ge ett hopp till den angivna rutinen. Om *rad* inte finns får man ett "Undefined line" ("Raden ej definierad") felmeddelande. En RESUME-sats behövs då för att återuppta programkörningen.

ON ERROR GOTO 0 stänger av felhanteringen. Alla fel som uppkommer efter en sådan sats kommer att ge felmeddelanden och avbryta programkörningen. ON ERROR GOTO 0 i en felhanteringsrutin gör att det uppkomna felets felmeddelande skrivs ut och programmet stannar. Det kan rekommenderas att en ON ERROR GOTO 0-sats utförs för fel som man inte har någon felhantering till.

Se även: RESUME, ERR, ERL

Exempel:

```
10 ON ERROR GOTO 900
900 IF (ERR = 230) AND (ERL=90) THEN PRINT "try again" : RESUME 80
```

ON *uttryck* {GOSUB}{GOTO} *adresslista*

Ger hopp till en av flera angivna radnummer eller radadresser beroende på värdet av *uttryck.*

Värdet av *uttryck* avgör vilken adress i *adresslista* som skall användas. Om värdet inte är ett heltal så avrundas talet.

adresslista är en uppräkning av radnummer eller radadresser dit hopp sker beroende på värdet av *uttryck*. Om t ex uttryckets värde är 3 så kommer tredje adressen i adresslistan att bestämma vart hoppet skall

ske.

I ON...GOSUB måste varje rad som är uppräknad i *radlista* vara den första raden i en relaterad subrutin.

Om värdet av *uttryck* är 0 eller större än antalet poster i *adresslista* (men mindre än eller lika med 255) så fortsätter programkörningen med nästa rad. Om värdet är negativt eller större än 255 får man ett "Illegal function call" ("Felaktigt funktionsanrop") felmeddelande.

Exempel:

```
'This program illustrates the use of the
'ON...GOSUB Statement
START:
INPUT "Enter your choice number (1...3) ? ",CHOICE%
IF CHOICE% < 1 OR CHOICE% >3 THEN GOTO START:
ON CHOICE% GOSUB SUB1,SUB2,SUB3
END
SUB1:
   PRINT "SUBROUTINE ONE"
   RETURN
SUB2:
   PRINT "SUBROUTINE TWO"
   RETURN
SUB3:
   PRINT "SUBROUTINE TREE"
   RETURN
```

ON MENU GOSUB *{adress}{0}*

Ger hopp till angiven rutin så fort funktionen MENU(0) returnerar ett värde skilt från 0, dvs så fort användaren valt en menypost.

adress är en radadress eller ett radnummer i den subrutin till vilket ett eventuellt hopp sker. GOSUB 0 stänger av menyhanteringen, dvs ON MENU kommer inte ha någon funktion förrän en MENU ON-sats utförts.

Se även: MENU, "Avbrottshantering" i kapitel 6.

ON MOUSE GOSUB *{adress}{0}*

Ger hopp till specificerad rutin så fort musens valknapp (den vänstra knappen) trycks ned.

adress är en radadress eller ett radnummer i den subrutin till vilket ett eventuellt hopp sker. GOSUB 0 stänger av menyhanteringen, dvs ON MOUSE kommer inte ha någon funktion förrän en "MOUSE ON"-sats utförts.

Se även: MOUSE, MOUSE(), "Avbrottshantering" i kapitel 6.

ON TIMER*(n)* GOSUB *adress*
ON TIMER GOSUB 0

Ger hopp till specificerad rutin när en viss tid har förflutit.

Hopp till specificerad rutin kommer att ske var *n*:te sekund. *n* är större än 0 och mindre än 86400 (antalet sekunder på ett dygn).

adress är en radadress eller ett radnummer i den subrutin till vilket ett eventuellt hopp sker.

GOSUB 0 stänger av menyhanteringen, dvs ON TIMER kommer inte ha någon funktion förrän en TIMER ON-sats utförts.

Se även: TIMER, "Avbrottshantering" i kapitel 6, "Avancerad programmering".

OPEN

Syntax 1: OPEN operationsläge,[#]filnummer, filspec[,filbufferstorlek]

Syntax 2: OPEN filespec[FOR operationsläge] AS
[#]filnummer[LEN=filbufferstorlek]

Allmänt om båda syntaxerna:

Öppnar en fil som finns på en skiva eller på någon annan utrustning för läsning eller skrivning.

Associerar ett *filnummer* med ett filnamn.

En fil måste alltid öppnas innan man kan utföra någon I/O-operationer

med den. OPEN reserverar utrymme åt en buffert för filen eller utrustningen och bestämmer vilken typ av operationer som skall användas.

filnummer är ett heltalsuttryck vars värde är från 1 till 255. Numret är associerat med filen så länge den är öppen och används av andra satser för att referera till filen.

filspec är ett stränguttryck som innehåller filens namn och, valfritt, namnet på skivvolymen, skivenheten eller utrustningen.

filbufferstorlek kan högst vara 32767 bitgrupper. Om *filbufferstorlek* inte anges så antas 128 bitgrupper. Vid direktåtkomstfiler så anger *filbufferstorlek* postlängden (antal tecken i en post). Vid sekventiella filer så behöver *filbufferstorlek* inte ange individuella posters storlek då sekventiella filer kan ha poster med olika storlek. *filbufferstorlek* anger istället antal tecken som lagras i filbufferten innan de skrivs till skivan eller innan det läses in nya tecken från skivan. Ju större buffert, desto större I/O-hastighet men mindre utrymme för Amiga Basic.

Speciellt syntax 1:

Vid OPEN:s första syntax är *operationsläge* ett stränguttryck vars första tecken är något av följande:

O	Anger sekventiellt skrivläge.
I	Anger sekventiellt läsläge.
R	Anger direktåtkomst läs/skrivläge.
A	Anger sekventiellt lägg-till-läge.

Speciellt syntax 2:

Vid OPEN:s andra syntax så är *operationsläge* ett av följande nyckelord:

OUTPUT	Anger sekventiellt skrivläge.
INPUT	Anger sekventiellt läsläge.
APPEND	Anger sekventiellt lägg-till-läge. Filpekaren sätts till filslutet och PRINT# och WRITE#-satser kommer att addera poster till filens slut.

Om *operationsläge* utelämnas i syntax 2 så antas direktåtkomstläge.

Exempel:

```
OPEN "ball" FOR INPUT AS 1
OPEN FileNameA$ FOR INPUT AS 2
OPEN FileNameB$ FOR OUTPUT AS 3
```

OPTION BASE *n*

Definierar minsta möjliga värde (n) för index hos indexerade variabler.

Om *n* är 1 så är det minsta möjliga värdet 1, om n är 0 så är det minsta möjliga värdet 0. Förvalt värde är 0. Specificerar man ett annat värde än 0 och 1 så får man ett syntaxfel.

OPTION BASE måste utföras före någon indexerad variabel definieras eller används.

Exempel:

```
OPTION BASE 1
```

PAINT [STEP]*(x,y) [,målarfärgid [,gränslinjefärgid]]*

Målar en slutet område (t ex en cirkel) med angiven färg.

x och *y* är koordinaterna för en punkt i det slutna området.

STEP anger att x och y är relativa koordinater till senast angiven punkt, annars anger x och y en absolut punkt i resultatfönstret.

målarfärgid anger vilken färg som området skall målas med. Om man inte anger *målarfärgid* så används den förgrundsfärg som angivits i en COLOR-sats.

gränslinjefärgid anger vilken färg linjen som begränsar området skall ha. Utelämnar man denna parameter så får gränslinjen den färg som angivits med *målarfärgid*.

Både *målarfärgid* och *gränslinjefärgid* refererar till *färgid*-parametern i en PALETTE-sats.

Anmärkning: Man måste ange en typ från 16 till 31 i WINDOW-satsen som skapat fönstret med området som skall målas.

Se även: PATTERN, AREA, AREAFILL

Exempel:

```
hue=RND*3
CIRCLE (x,y),radius,hue
PAINT (x,y),hue
```

PALETTE *färgid, röd, grön, blå*

Definierar den färguppsättning (palett) som vissa andra instruktioner använder sig av.

färgid identifierar paletten, dvs andra instruktioner använder detta värde för att referera till paletten. *färgid* är ett tal från 0 till 31. Djup-parametern i SCREEN-instruktionen bestämmer hur många färger man kan använda genom att ange det största värdet *färgid* kan ha.

Anmärkning: Amiga-systemet använder sig av *färgid* 0, 1, 2 och 3; om man knyter en färg till någon av de siffrorna med en PALETTE-sats så kommer man att undertrycka systemets färgval. Amiga-systemet har följande färgval knutet till siffrorna 0, 1, 2 och 3:

0	blå
1	vit
2	svart
3	orange

Om man refererar till dessa nummer i satser som kräver en *färgid* så skall man tänka på att användaren kan ändra denna inställning med Preference-verktyget som finns på Workbenchen.

röd, grön, och *blå* innehåller värden från 0.00 till 1.00. De anger, i procent, mängden rött, blått och grönt i färgen. Tabellen nedan visar vilken färg olika blandningar ger upphov till.

Färg	Röd	Grön	Blå
Aquablått	0.00	0.93	0.87
Svart	0.00	0.00	0.00
Mörkblått	0.40	0.60	1.00
Ljusblått	0.47	0.87	1.00
Brun	0.80	0.60	1.00
Grå	0.73	0.73	0.73
Grön	0.33	0.87	1.00
Citrongrön	0.73	1.00	0.00
Orange	1.00	0.73	0.00
Lila	0.80	0.00	0.93
Röd (körsbär)	1.00	0.60	0.67
Röd (eld)	0.93	0.20	0.00
Beige	1.00	0.87	0.73
Violett	1.00	0.13	0.93
Vit	1.00	1.00	1.00
Gul	1.00	1.00	0.13

Exempel:

```
PALETTE 1,RND,RND,RND
PALETTE 2,RND,RND,RND
COLOR 1,2
```

PATTERN *[linjemönster] [områdemönster]*

linjemönster är ett heltalsuttryck som definierar ett 16-bitars filter som används för att bestämma utseendet på de linjer man ritar.

områdemönster är namnet på en indexerad variabel som innehåller mönstret. Den indexerade variabeln beskriver ett 16-bitars brett och N-bitars högt filter som används för att bestämma utseendet på det område som en polygon omger. N är antalet element i den indexerade variabeln och måste minst vara 2.

För mer information, se kapitlet "Graphics Support" i *Amiga ROM Kernel Manual*.

Se även: AREA och COLOR.

Exempel:

```
DIM AREA.PAT%(3)
AREA.PAT%(0)=&H5555
AREA.PAT%(1)=&HAAAA
AREA.PAT%(2)=&H5555
AREA.PAT%(3)=&HAAAA
PATTERN &HFFF,AREA.PAT%
```

PEEKL*(adress)*

Ger ett långt heltalsord (32-bitars ord) läst från minnet på adress *adress*.

adress är ett numeriskt uttryck från 0 till 16 777 216; det anger adressen i minnet varifrån ordet skall läsas. Det numeriska uttrycket måste vara ett jämnt nummer.

Se även: POKEL

PEEK*(adress)*

Ger ett heltal med längden en byte läst från minnet på adress *adress*.

Det returnerade värdet är ett heltal från 0 till 255.

adress måste vara ett tal från 0 till 16 777 215.

Se även: POKE

PEEKW*(adress)*

Ger ett kort heltalsord (16-bitars ord) läst från minnet på adress *adress*.

adress är ett numeriskt uttryck från 0 till 16 777 216; det anger adressen i minnet varifrån ordet skall läsas. Det numeriska uttrycket måste vara ett jämnt nummer.

Se även: POKEW

POINT*(x,y)*

Ger färgid för punkten x,y i det aktuella resultatfönstret.

x och *y* är koordinaterna i det aktuella resultatfönstret för det bildelement som skall undersökas. Funktionen returnerar ett nummer som refererar till *färgid* i en PALETTE-sats.

Koordinaten 0,0 anger punkten i övre vänstra hörnet av det aktuella resultatfönstret. Anger man en punkt utanför det aktuella resultatfönstret så returnerar funktionen värdet -1.

POKE I,J

Skriver en byte till minnet.

I och J är heltalsuttryck. Uttrycket I anger minnesadressen, J är en byte med ett värde från 0 till 255. Parametern I skall ha ett värde mellan 0 och 16 777 215.

Varning: Använd POKE försiktigt, felaktig användning kan få systemet att låsa sig. Man får då starta datorn på nytt.

Se även: PEEK, VARPTR

POKEL *adress, värde*

Skriver ett långt heltalsord till minnet på adress *adress*.

adress är ett numeriskt uttryck med ett värde från 0 till 16 777 216. Uttryckets värde måste vara ett jämnt nummer.

värde är ett numeriskt uttryck med ett värde från -2 147 483 648 till 2 147 483 648.

Använd POKEL försiktigt, felaktig användning kan få systemet att låsa sig. Man får då starta datorn på nytt.

Se även: PEEKL

POKEW *adress, värde*

Skriver ett kort heltalsord till minnet på adress *adress*.

adress är ett numeriskt uttryck med ett värde från 0 till 16 777 216. Uttryckets värde måste vara ett jämnt nummer.

värde är ett numeriskt uttryck med ett värde från -65 536 till 65 535; värden utanför detta område huggs av till 16 bitar och lagras sedan på angiven adress.

Använd POKEW försiktigt, felaktig användning kan få systemet att låsa sig. Man får då starta datorn på nytt.

Se även: PEEKW

POS*(x)*

Ger, approximativt, numret på den kolumn som pennan befinner sig i.

Vid uträkning av kolumnnumret används tecknet "0":s bredd i det aktuella resultatfönstrets teckenuppsättning.

Det returnerade värdet är större än eller lika med 0. Den horisontella parametern i LOCATE-instruktionen är motsatsen till POS.

Exempel:

```
Y = CSRLIN   ' GET CURRENT CURSOR LINE NUMBER (VERTICAL POSITION)
X = POS(0)   ' GET CURRENT CURSOR COLUMN NUMBER (HORIZONTAL POSITION)
LOCATE 20,1 ' PLACE CURSOR ON LINE 24, ROW 1 (BOTTOM OF SCREEN)
PRINT "THIS PRINTS AT LOCATION 20,1 (BOTTOM OF PAGE)"
LOCATE Y,X   ' PLACE CURSOR IN ORIGINAL LOCATION
PRINT "THIS PRINTS AT ORIGINAL LOCATION OF CURSOR"
```

PRESET [STEP]*(x,y) [,färgid]*

Färglägger en punkt i det aktuella resultatfönstret.

PRESET fungerar på samma sätt som PSET förutom att om man

utelämnar *färgid* så används bakgrundsfärgen.

x och *y* är bildelementets koordinater. "STEP" gör att koordinaterna blir relativa den senast angivna punkten. Om STEP utelämnas anger koordinaterna en absolut punkt.

färgid anger vilken färg som skall användas; den refererar till *färgid* i en PALETTE-sats.

Om man anger en punkt utanför det aktuella resultatfönstret så kommer inget att uföras och inget felmeddelande kommer att ges.

Syntaxen för STEP är:

STEP*(x-avstånd,y-avstånd)*

där *x-avstånd* anger avståndet från senast angivna x-koordinat, *y-avstånd* anger avståndet från senast angivna y-koordinat. Om senast angivna punkt är 10,10 så skulle STEP(10,0) referera till punkten 20,10.

Se även: PSET och COLOR.

PRINT *[uttryckslista]*

Skriver data på skärmen i det aktuella resultatfönstret.

Om *uttryckslista* utelämnas så skrivs en blankrad ut. Om *uttryckslista* används så kommer uttryckens värden att skrivas ut i det aktuella resultatfönstret. Uttrycken är numeriska uttryck eller stränguttryck. Strängkonstanter måste vara omgivna av citationstecken.

Positionering av utskriften

Den position som ett värde skrivs ut i avgörs av de tecken som används för att åtskilja uttrycken i *uttryckslista*. Ett kommatecken gör att efterföljande värde skrivs ut vid nästa tabulatorstop (som definieras med WIDTH). Ett semikolon gör att värdet skrivs direkt efter föregående värde. Mellanslag (ett eller flera) fungerar på samma sätt som semikolon.

Om *uttryckslista* avslutas med ett kommatecken eller ett semikolon så kommer nästa "PRINT" sats att skriva på samma rad. I alla andra fall så kommer <RETUR> att skrivas sist på raden. Om raden som skall skrivas

ut blir längre än radlängden (som den är definierad med WIDTH) så fortsätter utskriften på nästa rad.

Tal efterföljs alltid av ett blanktecken. Positiva tal föregås av ett blanktecken, negativa tal föregås av ett minustecken. Tal i enkel precision som kan representeras med 7 eller färre siffror skrivs på vanligt sätt, i annat fall skrivs de på exponentform. T ex så skrivs 1E-7 som .0000001 medan 1E-8 skrivs som 1E-8. Tal i dubbel precision som kan representeras med 16 eller färre siffror skrivs på vanligt sätt, i annat fall skrivs de på exponentform.

Anmärkning: Man kan använda frågetecken (?) istället för ordet PRINT i en PRINT-sats.

Se även: LPRINT

PRINT USING *stränguttryck;uttryckslista*

Skriver strängar eller tal på skärmen i ett format som man själv bestämmer.

stränguttryck är en textsträng eller en strängvariabel med speciella formateringstecken. Dessa formateringstecken bestämmer fältet och formatet av de strängar eller tal som skall skrivas ut. Om man vill att formateringstecknen (!, &, #, etc, beskrivs längre fram) bokstavligen skall skrivas ut så måste de föregås av ett understrykningstecken (_).

uttryckslista innehåller de stränguttryck eller numeriska uttryck som skall skrivas ut; uttrycken måste vara åtskilda från varandra med kommatecken eller semikolon.

Formaterad utskrift av strängar

Man använder formateringstecknen !, \\, och & för att formatera strängutskrifter.

!	Anger att bara det första tecknet i strängen skall skrivas ut.
n blanka\\	Representerar ett valfritt antal blanka tecken mellan två omvända snedstreck. Anger att 2+n tecken i strängen skall skrivas ut. Om man anger \\

så skrivs två tecken ut. För varje blanktecken du sätter in mellan snedstrecken så kommer ett tecken till att skrivas ut. T ex,

\ \

gör att tre tecken skrivs ut. Om man anger fler blanka än det finns tecken i strängen så kommer Amiga Basic att lägga till motsvarande blanktecken i början av strängen, dvs strängen kommer att skrivas ut vänsterjusterad. Om man anger färre blanka än antalet tecken i strängen så kommer de överskjutande tecknen i strängen att ignoreras.

&	Anger att hela strängen skall skrivas ut.

Formaterad utskrift av tal

Följande formateringstecken används vid utskrift av numeriska uttryck:

Tecken	Funktion
#	Anger antalet teckenpositioner som skall reserveras för siffror.
	Sätter in en decimalpunkt.
+	Sätter in ett plus eller minustecken.
-	Sätter in ett efterföljande minustecken vid negativa tal.
**	Fyller inledande blanka med asterisker.
$$	Skriver ett dollartecken precis till vänster om ett tal.
**$	Fyller inledande blanka med asterisker och sätter in ett dollartecken.
,	Skriver kommatecken till vänster om decimalpunkten.
^^^^	Anger exponentform.
_	Anger att bokstaven som följer skall skrivas ut som den ser ut.

Alla andra tecken skrivs ut som de ser ut.

T ex:

```
PRINT USING "BALANCE = $$####.##";balance
```

#

"#" anger antalet teckenpositioner som skall användas vid utskriften av talet. Har man angivit fler positioner än talet upptar så skrivs talet ut högerjusterat.

Man kan sätta in en decimalpunkt mellan "#" tecknen för att reservera ett antal positioner för talets heltalsdel och ett antal positioner för decimaldelen. Decimaldelen föregås av nollor om nödvändigt.

Exempel:

```
PRINT USING "##.##";.78
PRINT USING "##.##";10.2,5.3,.234
```

Följande tal skrivs ut på skärmen:

```
0.78
10.20 5.30 0.23
```

+

Ett plustecken i början eller i slutet av en formatsträng gör att talets tecken (plus eller minus) skrivs ut före eller efter talet.

-

Om formatsträngen avslutas med ett minustecken så kommer negativa tal att skrivas med minustecknet på höger sida.

Exempel:

```
PRINT USING "+##.##";-68.95, 2.4, -9
PRINT USING "##.##-";-68.95, 22.499, -7
```

Följande skrivs ut:

```
-68.95 +2.40 -9.00
68.95 -22.45 7.00-
```

**

Två asterisker i början av formatsträngen gör att inledande blanka fylls med asterisker. Den andra asterisken reserverar även plats för två eller flera siffror. Satsen:

```
PRINT USING "**#.##"; 12.39, -0.9, 765.1
```

skriver följande:

```
*12.39*-0.90765.10
```

$$

Två dollartecken gör att ett dollartecken skrivs ut direkt till vänster om talen. De reserverar också utrymme för två tecken till, ett är till för dollartecknet. Man kan inte använda exponentform med "$$". Negativa tal kan bara användas om minustecknet sätts på höger sida om talet.

Satsen:

```
PRINT USING "$$###.##"; 456.78, 9.3
```

skriver följande:

```
$456.78 $9.30
```

****$**

"**$" är en kombination av det två föregående tecknen. Inledande blanka fylls med asterisker och ett dollartecken skrivs ut före talet. Tre teckenpositioner till reserveras; en används av dollartecknet. Man kan inte använda exponentform med "**$". Vid negativa tal hamnar minustecknet just till vänster om dollartecknet.

Exempel:

```
PRINT USING "**$##.##"; 2.34, 999.9
```

Följande skrivs ut på skärmen:

```
***$2.34*$999.90
```

Om man sätter in ett kommatecken till vänster om decimalpunkten i formatsträngen så skrivs ett kommatecken ut till vänster om var tredje siffra. Detta påverkar inte utskriften av siffrorna till höger om

decimalpunkten. Ett kommatecken i slutet av formatsträngen skrivs ut. Kommatecken reserverar en teckenposition. Kommatecken har ingen funktion tillsammans med "^^^^".

Exempel:

```
PRINT USING "####,.##"; 1234.5
PRINT USING "####.##,"; 1234.5
```

Följande skrivs ut:

```
1,234.50
1234.50,
```

_

Man använder understrykningstecknet för att kunna skriva ut formateringstecknen som de ser ut.

Exempel:

```
PRINT USING "_!##.##_!";12.34
PRINT USING "_?##.##_?";12.34
```

Följande skrivs ut:

```
!12.34!
?12.34?
```

^^^^

Genom att placera "^^^^" efter "#"-tecknet så skrivs talen ut på exponentform. Plats reserveras för "E+" eller "E-". Man kan samtidigt ange en decimalpunktsposition, siffrorna kommer då att vänsterjusteras på det sätt som behövs och exponenten kommer att justeras. Om man inte använder sig av avslutande "+" eller "-" eller ett inledande "+" så kommer ett blanktecken eller ett minustecken att skrivas ut till vänster om decimalpunkten.

Exempel:

```
PRINT USING "##.##^^^^";234.56
PRINT USING ".####^^^^";888888
PRINT USING "+.####^^^^";123
```

Följande skrivs ut:

```
2.35E+02
.8889E+06
+.12E+03
```

% - spillindikatorn

Om ett tal är för stort för att få.plats i ett fält så skrivs ett procenttecken ut för att indikera att spill uppstått.

Exempel:

```
PRINT USING "##.##";987.654
```

Följande skrivs ut:

```
%987.65
```

Om antalet siffror överstiger 24 så får man felmeddelandet "Illegal function call" ("Felaktigt funktionsanrop").

PRINT# *filnummer,* [USING *stränguttryck;*] *uttryckslista*

Skriver data i en sekventiell fil.

filnummer refererar till det nummer som användes då filen öppnades. *stränguttryck* består av de formateringstecken som beskrivs under PRINT USING. *uttryckslista* består av de numeriska uttryck eller stränguttryck som skall skrivas till filen.

PRINT# skriver data i filen på samma sätt som PRINT skulle ha skrivit det på skärmen. Datat komprimeras sålunda inte.

Man åtskiljer numeriska uttryck som i följande exempel:

```
PRINT #1,A;B;C;X;Y;Z
```

Om man åtskiljer uttrycken med kommatecken så kommer extra blanktecken att skrivas till filen.

Stränguttryck måste åtskiljas med semikolon *och* ett annat tecken. Följande exempel illustrerar vad som händer om man **bara** använder

semikolon som åtskiljningstecken:

```
A$="CAMERA"
B$="93604-1"
PRINT# 1,A$;B$
```

Följande sträng kommer att finnas i filen:

```
CAMERA93604-1
```

A$ och B$ kommer att lagras som en enda sträng!

Man kan lösa detta problem genom att använda ett kommatecken som åtskiljningstecken:

```
PRINT# 1,A$;",";B$
```

Nu skrivs följande till filen:

```
CAMERA,93604-1
```

De två variabelvärdena är åtskilda med ett kommatecken.

Strängar som innehåller kommatecken, semikolon, inledande blanktecken eller <RETUR>-tecken måste omges med dubbla citationstecken. Man använder CHR$(34) för detta ändamål (se CHR$ för mer information om funktionens arbetssätt).

Exempel:

```
A$="CAMERA, AUTOMATIC"
B$="93604-1"
PRINT #1,A$;",";B$
```

Följande skrivs till filen:

```
CAMERA, AUTOMATIC,93604-1
```

Om man läser data från denna fil med satsen:

```
INPUT #1,A$,B$
```

så får man följande tilldelningar:

```
A$="CAMERA"
B$="AUTOMATIC"
```

Som du ser så stämmer detta inte med den ursprungliga tilldelningen.

Man löser problemet på följande sätt:

```
PRINT #1,CHR$(34);A$;CHR$(34);",";B$
```

Med hjälp av CHR$(34) har man satt dubbla citationstecken runt A$. Om man nu läser data från filen med satsen:

```
INPUT #1,A$,B$
```

så får man följande tilldelningar:

```
A$="CAMERA, AUTOMATIC"
B$="93604-1"
```

vilket ju är korrekt!

Man kan också använda sig av PRINT# USING för att formatera strängar som skall skrivas till en fil.

Exempel:

```
PRINT #1,USING "$$###.##,";J;K;L
```

Se även: WRITE

PSET [STEP] *(x,y) [,färgid]*

Färglägger en punkt i det aktuella resultatfönstret.

x och *y* är bildelementets koordinater. STEP gör att koordinaterna blir relativa den senast angivna punkten. Om STEP utelämnas anger koordinaterna en absolut punkt.

färgid anger vilken färg som skall användas; den refererar till *färgid* i en PALETTE-sats.

Om man anger en punkt utanför det aktuella resultatfönstret så kommer inget att uföras och inget felmeddelande ges.

Syntax för STEP är:

STEP*(x-avstånd,y-avstånd)*

där *x-avstånd* anger avståndet från senast angivna x-koordinat, *y-avstånd* anger avståndet från senast angivna y-koordinat. Om senast angivna punkt är 10,10 så refererar STEP(10,0) till punkten 20,10.

Se även: PRESET och COLOR.

Exempel:

```
'Draw a thousand stars in random locations
FOR I = 1 TO 1000
  x = INT(RND*620)
  y = INT(RND*200)
  PSET(x,y)
NEXT I
```

PTAB(X)

Flyttar skrivpositionen till bildelement X.

PTAB fungerar på samma sätt som TAB förutom att X anger positionen räknat i bildelement i stället för i tecken. Om aktuell skrivposition är till höger om X så återgår skrivpositionen till X. 0 är positionen längst till vänster. X måste vara från 0 till 32 767. PTAB kan bara användas i PRINT-satser. När PTAB används så kommer PRINT inte att framtvinga <RETUR>.

PUT *[#] filnummer [,postnummer]* (1)
PUT [STEP] *(x,y),variabelnamn [(index[,index...])] [,aktionsverb]* (2)

De två syntaxerna visar två olika användningsområden för PUT:

1. Fil-PUT: Skriver en post som finns i direktåtkomstbufferten till en direktåtkomstfil.

2. Skärm-PUT: Ritar en bild på skärmen (som tidigare hämtats med GET).

Fil-PUT

filnummer refererar till det filnummer som filen öppnades med. Om *postnummer* utelämnas så antas det postnummer som står på tur (det efter senaste PUT). Postnumret måste vara mellan 1 och 16 777 215.

Man kan använda PRINT#, PRINT# USING, och WRITE# för att skriva tecken till direktåtkomstbufferten innan PUT-satsen utförs, men oftast så använder man FIELD tillsammans med RSET eller LSET för att fylla bufferten.

Om man använder WRITE# så fylls bufferten med blanktecken fram till <RETUR>-tecknet. Försöker man läsa eller skriva förbi buffertslutet, får man ett "Field overflow" ("Fältspill") felmeddelande.

Skärm-PUT

Skärm-PUT används för att rita en bild på skärmen. Koordinaten *x1,y1* anger var övre vänstra hörnet på den rektangulära bilden skall placeras i det aktuella resultatfönstret.

variabelnamn är namnet på den indexerade variabel som innehåller bilden. Se GET för utförligare förklaring.

Med hjälp av *index* kan man lagra flera bilder i samma variabel. Denna variabel kan man sedan uttnyttja för att snabbt växla mellan olika bilder.

aktionsverb är en av följande: PSET, PRESET, AND, OR, XOR. Om *aktionsverb* utelämnas så antas XOR. *aktionsverb* avgör hur bilden som finns på skärmen och bilden som skall ritas skall påverka varandra.

Se även: GET, PSET, PRINT, WRITE, FIELD, LSET, RSET

Exempel:

```
PUT (0,0),BobArray,PSET
```

RANDOMIZE *[uttryck]* | [TIMER]

Ger en ny följd av slumptal med *uttryck* som första tal. *uttryck* är ett tal mellan -32 768 och 32 767 eller, om man väljer TIMER, TIMER:s värde. Om man utelämnar *uttryck* så kommer Amiga Basic att stanna programkörningen och fråga efter ett värde genom att skriva:

```
Random Number Seed (-32768 to 32767)?
```

Om man inte använder sig av RANDOMIZE så kommer RND-funktionen att ge samma sekvens av tal varje gång programmet körs.

Se även: RND

READ *variabellista*

Läser värden från DATA-satser och lagrar dem i variabler.

En READ-sats måste användas tillsammans med DATA-satser. Värdena som READ läser lagras i variablerna som anges i *variabellistan*. Dessa variabler kan vara numeriska variabler eller strängvariabler och måste överensstämma med datatypen på de värden som läses från DATA-satserna. Om typerna inte överensstämmer så får man ett "Syntax error" ("Syntaxfel") felmeddelande.

Man kan med en ensam READ-sats läsa en eller flera DATA-satser eller med flera READ-satser läsa samma DATA-sats. DATA-satserna läses i den följd de kommer i programmet. Om antalet variabler i *variabellistan* är större än antalet element i DATA-satserna så får man ett "Out of data" ("Slut på data") felmeddelande. Om antalet angivna variabler är färre än antalet element i DATA-satserna så kommer READ-satser längre fram att börja läsa just efter det element som den den föregående READ-satsen läste. Om man vill läsa DATA-satserna från början använder man RESTORE.

Se även: DATA, RESTORE

Exempel:

```
DIM CF(19)
FOR I=1 TO 19
    READ CF(I)
    PRINT CF(I)
NEXT I
DATA  0,2,4,5,7,9,11,0,1,-1, 0,0,0,0,0,0, -12,12,0
```

REM *kommentar*

Används för att skriva kommentarer i program.

REM-satser påverkar inte programmets funktion på något sätt men syns vid programlistning.

Man får hoppa till REM-satser med GOTO- eller GOSUB-satser.

Man kan skriva in en kommentar i slutet av en rad genom att börja kommentaren med en apostrof (') istället för med REM.

Varning

DATA-satser behandlar REM som data. Man kan sålunda inte ha kommentarer i slutet av DATA-satser.

RESTORE *[rad]*

Anger från vilken rad DATA-satser skall läsas.

Om man utelämnar *rad* så kommer DATA-satserna att läsas från början. Om *rad* är angiven kommer data att börja läsas från den angivna DATA-satsen.

Exempel:

```
MainLoop:
  SOUND RESUME
  RESTORE Song
  GOSUB PlaySong
  GOTO MainLoop
     :
     :
     :
Song:
DATA 1,3,3,3
DATA l2g>ge, l2p2de, l2p2l6g3f#g3a, l6p6gab>dcced
```

RESUME [{0}{NEXT}*{rad}*]

Återupptar programkörning efter att ett fel har blivit åtgärdat. Fyra syntaxer finns:

RESUME eller RESUME 0	Programkörningen återupptas vid den sats som orsakade felet.
RESUME NEXT	Programkörningen återupptas på raden efter satsen som orsakade felet.
RESUME *rad*	Programkörningen återupptas på rad *rad.*

Om en RESUME-sats utförs även om inget fel har uppstått så får man ett "RESUME without error" ("RESUME utan fel") felmeddelande. Man måste sålunda alltid placera RESUME-satser i felhanteringsrutiner.

RETURN *[rad]*

Anger slutet på en subrutin och att återhopp skall ske.

rad fungerar på samma sätt som i en GOTO-sats. Om *rad* utesluts så sker återhopp till satsen efter anropande GOSUB-sats.

rad specifikationen gör RETURN-instruktionen mer flexibel. Är man oförsiktig med användningen av *rad* så kan man få svårfunna programfel.

Exempel:

```
5 MOUSE ON
10 ON MOUSE GOSUB 1000
20 FOR I=1 TO 10
30 PRINT I
40 NEXT I
50 REM NEXT PROGRAM LINE
   :
   :
200 REM PROGRAM RESUMES HERE
   :
   :
1000 'FIRST LINE OF SUBROUTINE
   :
   :
1050 RETURN 200
```

Om användaren trycker på musens valknapp då programmet befinner sig i FOR...NEXT-slingan så kommer ett hopp att ske till subrutinen, men när återhopp sker kommer det att ske till rad 200 som är utanför FOR...NEXT-slingan. FOR-satsen är fortfarande aktiv så man kommer att få ett "FOR without NEXT" felmeddelande.

Se även: GOSUB

**RIGHT$**(X$,I)

Ger I tecken från höger ur strängen X$.

Om I är större än eller lika med antalet tecken i X$ så returneras hela X$. Om I=0 så returneras en tom sträng. I är mellan 0 och 32767.

Se även: LEFT$, MID$

Exempel:

```
'THIS ROUTINE EXTRACTS THE ADDRESS a: FROM STRING RECORD$
'
RECORD$ = "n:JOHN JONES ss:5349 12 99 a:3633 6TH ST WACO,TX"
LENGTH = LEN(RECORD$)               'DETERMINE LENGTH OF RECORD
OFFSET = INSTR(RECORD$,"a:")        'FIND START OF ADDRESS a:
RIGHTCHAR = LENGTH - OFFSET - 1
ADDRESS$ = RIGHT$(RECORD$,RIGHTCHAR) 'EXTRACT ADDRESS FROM RECORD$
PRINT ADDRESS$
```

Följande visas på skärmen:

```
3633 6TH ST WACO,TX
```

RND[(X)]

Ger ett slumptal mellan 0 och 1.

RND ger samma sekvens av slumptal om man inte använder sig av RANDOMIZE.

* X<0 startar om den givna slumptalssekvensen.

* X>0 eller inget X ger nästa tal i slumptalssekvensen.

* X=0 repeterar det senaste genererade slumptalet.

Se även: RANDOMIZE

Exempel:

```
FOR I = 1 TO 40
   X = INT(RND*620)    'SET HORIZONTAL LOCATION OF CENTER
   Y = INT(RND*200)    'SET VERTICAL LOCATION OF CENTER
   RADIUS = 40*RND     'SET A RANDOM RADIUS
   CIRCLE (X,Y),RADIUS 'DRAW A CIRCLE
NEXT I
```

RSET *strängvariabel=stränguttryck*

Flyttar data från minnet till en direktåtkomstbuffert och förbereder på så sätt en PUT-sats.

Fungerar på samma sätt som LSET, men där LSET vänsterjusterar så högerjusterar RSET.

Se även: LSET

RUN *[rad][filnamn*[R]*]*

Startar det program som finns i datorns minne. Om det inte finns något program i minnet så kommer inget att hända.

"Start" från Run-menyn har samma funktion som RUN utan parametrar.

Anger man tillvalet *rad* så kommer programkörningen att påbörjas från raden som har det radnummer man angivit. Om radnumret inte finns får man ett "Undefined label" ("Odefinierad adress") felmeddelande.

rad kan inte vara en radadress. Man får ett "Type mismatch" felmeddelande om man använder radadresser tillsammans med RUN.

Varning: När man använder RUN *rad* så kan man råka hoppa över definitioner som t ex DIM, DEFINT, DEFSNG, DEFDBL, DEF FN, som finns på rader i början på programmet. Detta kan leda till fel längre fram i programmet.

Anger man tillvalet *filnamn* så kommer det namngivna programmet att läsas in och köras. Filnamnet måste vara omgivet av citationstecken och ha samma namn som programmet sparades under.

I *filnamn* kan man också ange på vilken skivstation, skivvolym och i vilken katalog som Amiga Basic skall söka efter filen.

Exempel:

```
RUN "basicdemos/picture"
```

Amiga Basic kommer i ovanstående exempel att leta efter filen picture.bas i katalogen basicdemos.

Om det finns ett program i minnet när man utför RUN *filnamn* så öppnas ett frågefönster som ger en möjlighet att spara programmet i minnet innan det nya programmet läses in.

RUN stänger alla öppna filer och rensar minnet från tidigare innehåll innan ett nytt program läses in och körs. Om man använder sig av R-tillvalet så lämnas dock alla datafiler öppna. Man kan på så sätt föra över information från ett program till ett annat.

SADD*(stränguttryck)*

Ger minnesadressen för den aktuella strängens första byte.

Det värdet som returneras är aktuellt fram till dess att en strängvariabel tilldelas ett värde. En tilldelning kan få redan lagrade strängar att flytta plats i minnet. SADD används oftast för att skicka minnesadressen för en sträng till en maskinspråksrutin.

Undvik att använda VARPTR(stränguttryck) då dess syntax kanske kommer att ändras i framtida versioner av Amiga Basic.

Se även: VARPTR

Exempel:

```
CALL Prompt(SADD("How many"+CHR$(0)))
```

SAVE [*filnamn*[,{A}{P}{C}]]

Sparar ett program som finns i minnet på skiva.

filnamn är en sträng omgiven av citationstecken. Om filen redan finns på skivan så kommer den att bli överskriven. Om man inte anger *filnamn* så öppnas ett frågefönster som frågar efter ett filnamn.

Kommandot har tre tillval:

A Filen sparas i ASCII-format. Tar mer plats men vissa kommandon och instruktioner kan bara arbeta med ASCII-filer.

P Filen sparas i skyddad form, dvs filen kommer framöver inte gå att lista eller ändra i.

B Komprimerat binärt format. Tar minst plats och är förvalt värde.

I *filnamn* kan man också ange vilken skivstation eller vilken skivvolym som skall användas. Om man t ex vill spara filen "test" på skivan i skivstation 1 så skriver man:

```
SAVE "df1:test"
```

För att spara "test" i katalogen "datafiler" på aktuell skiva skriver man:

```
SAVE "datafiler/test"
```

Mer information om filspecifikationer kan man hitta i *AmigaDOS User's Manual*.

SAY *"sträng"*[,attributvariabel]

Översätter en sträng till ljud.

sträng består av fonetiska koder. *attributvariabel* är en indexerad heltalsvariabel med minst 9 element. I den lagrar man talvärden som ger ljudet olika egenskaper.

Man kan framställa fonem från engelska ord med hjälp av TRANSLATE-funktionen. I appendix H går vi igenom hur man på manuell väg konstruerar fonem.

Tabellen nedan förklarar de olika värden man kan ange i *attributvariabel*. Om man inte anger någon *attributvariabel* så antas de förvalda värden som är angivna i tabellen.

Attribut	**Element**	**Förklaring**
Frekvens (ton)	0	Anger basfrekvensen för ljudet i Hertz. Ange ett värde mellan 65 och 320. Förvalt värde är 110 (normal mansröst).
Modulation	1	Röstens modulering: Välj mellan två värden: 0 - Modulering och betoning av stavelser. Förvalt värde. 1 - Monotont ("robotspråk").
Hastighet	2	Anger röstens hastighet i ord per minut. Ange ett värde mellan 40 och

		400. Förvalt värde: 150.
Kön	3	Man kan välja mellan mansröst (0) och kvinnoröst (1). Mansrösten är förvalt värde.
Avstämning	4	Röstläge. Ange ett värde mellan 5000 och 28000, förvalt värde 22000
Volym	5	Ljudvolym. Ange ett värde mellan 0 (inget ljud alls) och 64 (högsta ljudvolym). Förvalt värde är 64.
Kanal	6	Anger vilken eller vilka kanaler som ljudet skall skickas ut på. Kanal 0 och 3 går till vänstra ljudutgången, kanal 1 och 2 går till högra. Ange någon av de koder som finns i Kanalkodtabellen längre fram i detta avsnitt. Förvalt värde är 10.
Talläge	7	Följande tallägen finns: 0 - Synkront tal. Den aktuella SAY-satsen fullbordas innan någon ny sats utförs. Detta är förvalt värde. 1 - Asynkront tal. Amiga Basic börjar utföra SAY-satsen men börjar sedan direkt med efterföljande satser.
Kontroll	8	Kontroll av talsyntesatorn. Denna parameter anger hur flera SAY-satser skall behandlas vid asynkront tal, dvs när talläge=1. Ange något av följande värden: 0 - SAY-satserna bearbetas i turordning och varje sats avslutas innan en ny påbörjas. 1 - Avbryter talsynteseringen. 2 - Avbryter pågående SAY-sats och påbörjar nästa SAY-sats.

Kanalkodtabell

Värde	Kanal(er)
0	0
1	1
2	2
3	3
4	0 och 1
5	0 och 2
6	3 och 1
7	3 och 2
8	Någon tillgänglig vänsterkanal
9	Någon tillgänglig högerkanal
10	Något tillgängligt vänster/höger kanalpar (förvalt värde)
11	Någon tillgänglig kanal

Se även: TRANSLATE$

Exempel:

```
FOR J = 0 TO 8:  READ HOW%(J): NEXT J
TEXT$ = "DHIHS IHZ YOHR (AHMIY5GAH PER5SINUL KUMPYUW5TER) SPIY4KIHNX."
SAY TEXT$,HOW%
SAY TRANSLATE$ ("Hi there, how are you?")
DATA 110,0,250,0,22200,64,10,0,0
```

SCREEN {CLOSE} *skärmid {, bredd, höjd, djup, läge}*

Definierar skärmens storlek, antal färger och upplösning. SCREEN CLOSE stänger skärmen.

SCREEN reserverar minne för skärmens innehåll, SCREEN CLOSE frigör minnet.

skärmid är ett nummer mellan 1 och 4 som identifierar skärmen; WINDOW-satser använder sig av samma nummer för att ange på vilken skärm ett fönster skall öppnas.

bredd anger bredden på skärmen i bildelement. Ange ett värde från 1 till 640.

höjd anger höjden på skärmen. Ange ett värde från 1 till 400.

djup avgör hur många färger man kan använda på skärmen. Beroende på värdet kan man använda olika antal färger:

Värde	Antal färger
1	2
2	4
3	8
4	16
5	32

läge avgör bildelementens bredd på skärmen: 320 bildelement per vågrät linje vid lågupplösning, 640 bildelement per linje vid högupplösning. *läge* avgör också om de vågräta linjerna skall vara enkla eller dubbla. Lågupplösningen används när man använder en TV-apparat som bildskärm, högupplösning används med RGB-skärmar och högupplösande monokroma skärmar.

Om man använder dubbla vågräta linjer så behövs det bara 200 linjer för att fylla skärmen, med enkla linjer behövs det 400.

Tabellen nedan anger de värden som *läge* kan anta och vilken skärmtyp som blir resultatet:

Läge	Skärmtyp
1	Lågupplösning, dubbla vågräta linjer.
2	Högupplösning, dubbla vågräta linjer.
3	Lågupplösning, enkla vågräta linjer.
4	Högupplösning, enkla vågräta linjer.

Exempel:

```
SCREEN 1,320,200,5,1
WINDOW 2,"Lines",(10,10)-(270,170),15,1
```

SCROLL *rektangel, delta-x, delta-y*

Rullar ett område i det aktuella resultatfönstret.

rektangel definierar området som skall rullas och anges på formen

(x1,y1)-(x2,y2).

delta-x anger antalet bildelement som området skall rulla till höger. Om *delta-x* är negativt så rullar området åt vänster.

delta-y anger antalet bildelement som området skall rulla nedåt. Ett negativt värde gör att området rullar uppåt.

SCROLL fungerar bäst om bilden som skall rullas är mindre än det definierade området och om området som berörs inte har någon bakgrund.

SGN(X)

Avgör om ett tal är positivt, negativt eller 0.

Om X>0 så ger SGN(X) 1

Om X=0 så ger SGN(X) 0

Om X<0 så ger SGN(X) -1

Exempel:

```
PRINT SGN(-299)
PRINT SGN(499-499)
PRINT SGN(8722)
```

Följande visas på skärmen:

```
-1
0
1
```

SHARED *variabellista*

Variablerna i *variabellista* kommer att delas mellen ett underprogram och huvudprogrammet. Det betyder att variabler som har samma namn i

huvudprogrammet och underprogrammet kommer att innehålla samma värde.

variabellista är en uppräkning av de variabler som skall delas mellan underprogrammet och huvudprogrammet. Variablerna avskiljs med kommatecken. Indexerade variabler måste följas av parenteser, "( )".

Om en delad variabels värde förändras i underprogrammet så förändras motsvarande variabel i huvudprogrammet och vice versa.

SHARED-satser får bara användas i underprogram. Man kan ha flera SHARED-satser för olika variabler på samma sätt som kan ha flera DIM-satser för olika variabler.

SHARED-satserna bör skrivas först i underprogrammet; programmet blir då lättare att förstå.

Se även: DIM SHARED

SIN(X)

Ger sinusvärdet av X där X är i radianer.

Funktionen arbetar med tal i enkel eller dubbel precision beroende på om X är i enkel eller dubbel precision.

Se även: COS, TAN

Exempel:

```
PRINT "SINE OF 1 IS " SIN(1)
PRINT "SINE OF 100 IS " SIN(100)
PRINT "SINE OF 1000 IS " SIN(1000)
```

Följande visas på skärmen:

```
SINE OF 1 IS .841471
SINE OF 100 IS -.50663657
SINE OF 1000 IS .8268796
```

SLEEP

Stannar programkörningen tills en händelse inträffar, t ex klickning av musens valknapp, tangentnedtryckning, objektkollision, menyval, tidshändelse mm.

Exempel:

```
LOOP:
  I$=INKEY$
  IF I$="X" THEN STOP
  SLEEP
GOTO LOOP
```

SOUND *{frekvens, längd [,[volym][,kanal]]}*{WAIT}{RESUME}

Framställer ljud, upprättar en kö av ljud (WAIT) eller spelar upp en kö av ljud (RESUME).

SOUND WAIT gör att alla följande SOUND-satser ställs i en kö tills en SOUND RESUME-sats utförs. Man använder ofta funktionen för att synkronisera ljudet från Amigans fyra ljudkanaler. Kön kan inte bli hur lång som helst så om man ställer för många SOUND-satser i kö får man ett "Out of memory" ("slut på minnet") felmeddelande.

frekvens kan vara ett heltal eller ett decimaltal med enkel eller dubbel precision. *frekvens* är mellan 20 och 15000 Hertz, värden under eller över detta intervall avrundas till respektive gränsvärde.

Följande tabell visar fyra olika oktavers frekvens. Observera att en fördubbling av frekvensen ger en oktavs högre ton.

Not	Frekvens	Not	Frekvens
C	130.81	C*	523.25
D	146.83	D	587.33
E	164.81	E	659.26
F	174.61	F	701.00

G	196.00	G	783.99
A	220.00	A	880.00
B	246.94	B	993.00
C	261.63	C	1046.50
D	293.66	D	1174.70
E	329.63	E	1318.50
F	349.23	F	1396.90
G	392.00	G	1568.00
A	440.00	A	1760.00
B	493.88	B	1975.50

*Grundtonen C

längd är ett numeriskt uttryck mellan 0 och 77 och avgör hur länge ljudet varar. En sekund motsvaras av en *längd* på 18.2, dvs med en *längd* på 18.2 skulle man framställa ett ljud som varar i en sekund. Det maximala värdet 77 ger ett ljud som varar i 4.25 sekunder.

Följande tabell ger relationen mellan olika tempon och *längd*.

Tempo		**Slag per minut**	**Längd**
Mycket långsamt	Larghissimo	40-60	28.13-18.75
	Largo	60-66	18.75-17.05
	Larghetto		
	Grave		
	Lento		
	Adagio	66-76	17.05-14.8
Långsamt	Adagietto		
	Andante	76-108	14.8-10.42
Medium	Andantino		
	Moderato	108-120	10.42-9.38
Snabbt	Allegretto		
	Allegro	120-168	9.38-6.7
	Vivace		
	Veloce		
	Presto		
Mycket snabbt	Prestissimo	168-208	6.7-5.41

Tolken väntar alltid tills en SOUND-sats är färdig innan den utför nästa SOUND-sats.

volym är ett värde mellan 0 (lägsta volym) och 255 (högsta volym). Förvalt värde är 127.

kanal anger vilken av Amigans fyra ljudkanaler som skall användas. 0 eller 3 anger en av kanalerna till vänstra högtalaren, 1 eller 2 anger en av kanalema till högra högtalaren. Förvalt värde är 0.

Se även: WAVE

Exempel:

```
SOUND 440,20,100,0
```

SPACE$(X)

Ger en sträng som består av X blanktecken. Uttrycket X avrundas till ett heltal och måste vara mellan 0 och 32 767.

Se även: SPC

Exempel:

```
INDENT5$ = SPACE$(5)
INDENT10$ = SPACE$(10)
PRINT "Level 1 Outline Heading"
PRINT INDENT5$ "Level 2 Heading"
PRINT INDENT5$ "Level 2 Heading"
PRINT INDENT10$ "Level 3 Heading
PRINT INDENT10$ "Level 3 Heading"
PRINT "Level 1 Heading"
END
```

Följande visas på skärmen:

```
Level 1 Outline Heading
     Level 2 Heading
     Level 2 Heading
          Level 3 Heading
          Level 3 Heading
Level 1 Heading
```

SPC(I)

Ger I blanktecken när den används i PRINT- eller LPRINT-satser.

I är mellan 0 till 255.

<RETUR> utförs inte efter SPC(I).

Se även: PTAB, SPACE$, TAB

Exempel:

```
FOR I=1 TO 5
  PRINT SPC(I) "I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE"
NEXT I
```

Följande visas på skärmen:

```
I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
 I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
  I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
   I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
```

SQR(X)

Ger kvadratroten av X.

Uträkningen sker i enkel eller dubbel precision beroende på om X är i enkel eller dubbel precision.

Exempel:

```
PRINT "VALUE       SQUARE ROOT"
FOR I=1 TO 10
  PRINT I,SQR(I)
NEXT I
END
```

Följande visas på skärmen:

```
VALUE      SQUARE ROOT

1          1
2          1.414214
3          1.732051
4          2
5          2.236068
6          2.44949
7          2.645751
8          2.828427
9          3
10         3.162278
```

STICK*(n)*

Ger joystickens riktning.

n bestämmer vilken joystick (A eller B) och vilken koordinat (x eller y) som skall undersökas:

n	**Ger information om**
0	Joystick A:s x-koordinat
1	Joystick A:s y-koordinat
2	Joystick B:s x-koordinat
3	Joystick B:s y-koordinat

STICK ger ett av följande värden:

Värde	**Mening**
1	Rörelse uppåt eller till höger.
0	Ingen rörelse.
-1	Rörelse nedåt eller till vänster.

STRIG*(n)*

Ger joystickens aktuella status.

n avgör vad för information som returneras:

n	Returnerar
0	Ger 1 om joystick A:s knapp har tryckts ned sedan senaste STRIG(0) utfördes. Annars returneras 0.
1	Ger 1 om joystick A:s knapp är nedtryckt, annars returneras 0.
2	Ger 1 om joystick B:s knapp har tryckts ned sedan senaste STRIG(2) utfördes. Annars returneras 0.
3	Ger 1 om joystick B:s knapp är nedtryckt, annars returneras 0.

STOP

Avbryter programkörningen och återvänder till det interaktiva läget.

STOP kan användas var som helst i ett program. Man använder det ofta vid felsökning.

STOP stänger inte några filer.

Man kan återuppta programkörningen med CONT-kommandot.

Se även: CONT

STR$(X)

Ger en sträng som består av värdet på uttrycket X.

I strängen inleds positiva tal med ett blanktecken och negativa tal med ett minustecken.

Man använder inte STR$ för att omvandla tal till strängar då man arbetar med direktåtkomstfiler. Då använder man istället MKI$-, MKS$-, och MKD$-funktionerna.

Se även: VAL, MKI$, MKS$, MKD$

STRING$(I,J) (1)
STRING$(I,X$) (2)

Första syntaxen ger en sträng med längden I som består av tecken med ASCII-koden J.

Den andra syntaxen ger en sträng med längden I som består av det första tecknet i X$.

Exempel:

```
PRINT STRING$(10,"C")
PRINT STRING$(10,"#")
PRINT STRING$(10,37)
```

Följande visas på skärmen:

```
CCCCCCCCCC
##########
%%%%%%%%%%
```

SUB *underprogramnamn* *[(lista på formella parametrar)]* STATIC
EXIT SUB
END SUB

Startar, avslutar och hoppar ur ett underprogram.

underprogramnamn får vara 40 tecken långt och får inte användas i någon annan SUB-sats.

lista på formella parametrar innehåller två typer av variabler: enkla variabler och indexerade variabler. Indexnumret (som är tillval) efter indexerade variabler skall ange det högsta indexet som variabeln har. Variablerna separeras med kommatecken. Man får ha så många variabler som får plats på en rad.

STATIC gör att variablerna endast kan få och förändra värde i underprogrammet.

Satserna mellan SUB och END SUB utgör själva underprogrammet.

END SUB markerar slutet på underprogrammet. När denna sats utförs så kommer ett hopp till satsen efter den anropande satsen att utföras.

EXIT SUB ger hopp till satsen efter den anropande satsen.

Innan Amiga Basic kör ett program så kontrollerar den alla satser som hänger ihop med underprogrammen. Om tolken hittar ett fel så kommer programmet inte att köras. Felen kan inte hanteras med ON ERROR, de har heller inte några felkoder. Man får något av följande meddelande i det felfönster som öppnas:

`Tried to declare a SUB within a SUB.`
(Försök att deklarera ett underprogram i ett underprogram.)

`SUB already defined.`
(Underprogram redan definierat.)

`Missing STATIC in SUB statement.`
(STATIC saknas i SUB-sats.)

`EXIT SUB outside of a subprogram.`
(EXIT SUB-sats utanför underprogram.)

`END SUB outside of a subprogram.`
(END SUB-sats utanför underprogram.)

`SUB without an END SUB.`
(SUB-sats utan END SUB-sats.)

`SHARED outside of a subprogram.`
(SHARED-sats utanför underprogram.)

I kapitel 6 behandlas underprogrammens funktion och användningsområde utförligt.

Se även: CALL, SHARED

Exempel:

```
SUB Nextline(win) STATIC
  SHARED iDraw,iErase
  WINDOW OUTPUT win
  DrawLine iDraw,1
  DrawLine iErase,0
END SUB
```

SWAP *variabel1,variabel2*

Byter värde på två variabler.

Alla variabler av samma typ kan bytas. Om de är av olika typ får man ett "Type mismatch" ("Ej överensstämmande variabeltyper") felmeddelande.

Om *variabel2* inte är definierad när SWAP utförs så får man ett "Illegal function call" ("Felaktigt funktionsanrop") felmeddelande.

Exempel:

```
FIRST$ = "FRED"
LAST$ = "JONES"
PRINT FIRST$ SPC(1) LAST$
SWAP FIRST$,LAST$
PRINT FIRST$ SPC(1) LAST$
```

Följande visas på skärmen.

```
FRED JONES
JONES FRED
```

SYSTEM

Stänger alla öppna filer och återger kontrollen till Workbenchen.

SYSTEM kan användas både i program (som en instruktion) och i interaktivt läge i resultatfönstret (som ett kommando).

"Quit" från Workbenchens Project-meny ger samma resultat.

Både vid användningen av SYSTEM och "Quit" så kontrollerar tolken att programmet är sparat; är det inte sparat så öppnas ett frågefönster som ger möjlighet att spara programmet.

TAB(I)

Flyttar skrivpositionen till kolumn I.

Om den aktuella skrivpositionen är till höger om I så flyttar TAB skrivpositionen till kolumn I på nästa rad. I=1 är positionen längst till vänster och skärmbredden minus ett är positionen längst till höger. I måste dock vara mellan 1 och 155.

TAB kan bara användas tillsammans med PRINT och LPRINT.

<RETUR> utförs inte efter TAB(I).

Exempel:

```
PRINT " Name";TAB(16);"Amount Due"
PRINT TAB(2);"----";TAB(16);"----------"
FOR I%=1 TO 6
  READ A$,B
  PRINT " ";A$;TAB(18);B
NEXT I%
DATA "G. T. Jones",25,"A. Roos",1
DATA "G. Arrhenius",33,"B. Moore",99
DATA "T. Wernborg",100,"N. Styles",13.5
```

Följande skrivs ut på skärmen:

```
Name          Amount Due
----          ----------

G. T. Jones     25
A. Roos         1
G. Arrhenius    33
B. Moore        99
T. Wernborg     100
N. Styles       13.5
```

TAN(X)

Ger tangens av X där X är i radianer.

Uträkningen sker i enkel eller dubbel precision beroende på om X är i enkel eller dubbel precision.

Se även: COS, SIN

Exempel:

```
'Tangent request program
START:
INPUT "Enter a number ", NUMBER
PRINT "Tangent of "  NUMBER " is " TAN(NUMBER)
INPUT "If you have another number, enter y ", YORN$
IF YORN$ = "y" GOTO START
END
```

Följande skulle kunna visas på skärmen:

```
Enter a number 1.777
Tangent of 1.777 is -4.780646
If you have another number, enter y n
```

TIME$

Ger aktuell systemtid.

Funktionen returnerar en sträng på 8 tecken innehållande systemtiden på formen *tt:mm:ss* där *tt* anger timme (0 till 23), *mm* anger minut (00 till 59) och *ss* anger sekund (00 till 59).

Exempel:

```
PRINT TIME$      'PRINT CURRENT TIME IN COMPUTER
```

Följande skulle kunna visas på skärmen:

```
11:15:00
```

TIMER {ON}{OFF}{STOP}

Sätter på (ON), stänger av (OFF) och avbryter tillfälligt (STOP) avbrottshantering baserad på tid.

Med TIMER STOP så kommer inte ON TIMER...GOSUB-satsen att utföras. Om den angivna tiden överskrids så kommer ON TIMER GOSUB-satsen att utföras så fort en TIMER ON-sats utförs.

Se även: ON TIMER och kapitel 6.

Exempel:

```
ON TIMER(2) GOSUB TimeSlice  'Invoke TimeSlice every 2 seconds
TIMER ON
```

TIMER

Ger antalet sekunder som förflutit sedan midnatt.

TIMER kan också användas med RANDOMIZE för att generera olika följder av slumptal.

TRANSLATE$*("sträng")*

Översätter engelska ord till fonem som SAY kan använda för att framställa syntetiskt tal.

sträng är orden som skall översättas. Den fonemsträng som TRANSLATE$ returnerar kan inte överskrida 32 767 tecken.

Se även: SAY

Exempel:

```
A$=TRANSLATE$("There's no place like home.")
SAY(A$)
```

TRON
TROFF

Sätter på (TRON) och stänger av (TROFF) markering av aktuell sats vid programkörning.

"Trace On" och "Trace Off" på Run-menyn fungerar på samma sätt som TRON och TROFF.

TRON gör att den sats som just håller på att utföras markeras med en

orange rektangel i List-fönstret. List-fönstret måste vara öppet och synligt.

Markeringen upphör om man utför en TROFF-sats, väljer "Trace Off" från Run-menyn, eller om man utför ett NEW-kommando.

UBOUND*(variabelnamn[,dimension])*

Ger den övre indexgränsen för en dimension i en indexerad variabel.

Se LBOUND för information om både LBOUND och UBOUND.

UCASE$*(stränguttryck)*

Ger en sträng som endast består av versaler.

Strängen som funktionen returnerar är en kopia av *stränguttryck* förutom det att alla gemener omvandlas till motsvarande versaler.

Med UCASE$ kan man jämföra och sortera strängar som växelvis är inmatade med versaler och gemener. T ex, om man har en programrad:

```
INPUT "Vill du fortsätta", SVAR$
```

så kan användaren svara med "JA", "Ja", "ja", "J" eller "j". Istället för att testa alla dessa olika svar så kan man helt enkelt testa första bokstaven i den sträng som UCASE$(SVAR$) ger och se om den är lika med "J".

Om man skall sortera ett antal strängar i bokstavsordning, och dessa strängar inte är konsekvent inmatade med versaler eller gemener så kan man få problem. Om man t ex har de tre strängarna "arboga", "ARVIKA", och "BOXHOLM" så skulle datorn sortera dem i ordningen "ARVIKA", "BOXHOLM" och sist "arboga". Detta beror på att när strängar sorteras så jämför datorn deras ASCII-koder och stora A:s ASCII-kod är 65 medan lilla a:s ASCII-kod är 97. Om man istället sorterar på UCASE$-representationen av strängarna så kommer de att sorteras korrekt i bokstavsordning.

Exempel 1:

```
a$=UCASE(a$)
IF a$="Y" THEN Response=1
IF a$="N" THEN Response=2
IF a$="C" THEN Response=3
```

Exempel 2:

```
A$="ARVIKA"
B$="arboga"
C$="WaXhOlM"
PRINT A$,B$,C$
PRINT UCASE$(A$),UCASE$(B$),UCASE$(C$)
```

Notera skillnaden mellan resultatet av de två PRINT-satserna:

```
ARVIKA     arboga     WaXhOlM
ARVIKA     ARBOGA     WAXHOLM
```

VAL(X$)

Ger det numeriska värdet av X$.

VAL bortser från inledande blanktecken, tabulatortecken och radframmatningstecken i X$.

VAL används inte för att omvandla strängar i direktåtkomstfiler till tal. Man använder då CVI-, CVL-, CVS- och CVD-funktionerna.

Se även: STR$

VARPTR*(variabelnamn)*

Ger adressen för den första byten som används av variabeln *variabelnamn*. Ett värde måste lagras i variabeln innan VARPTR utförs, annars får man ett "Illegal function call" ("Felaktigt funktionsanrop") felmeddelande. Alla typer av variabler kan användas. Med strängvariabler så ger funktionen adressen till strängbeskrivningens första byte.

Adressen som returneras är mellan 0 och 16 777 215. För mer information, se appendix D, "Amiga Basic:s interna representation av tal".

Man använder ofta VARPTR för att skicka en enkel eller indexerad variabels adress till en maskinspråksrutin. För att en maskinspråksrutin skall komma åt alla element i en indexerad variabel så skickar man helt enkelt adressen till första variabelns element. T ex:

```
VARPTR(A(0)).
```

Anmärkning: Använd SADD-funktionen för att komma åt adressen till en strängvariabel.

Man måste tilldela alla element i en indexerad variabel ett värde innan man använder VARPTR då adresserna för de enskilda elementen förändras så fort ett värde tilldelas något av elementen.

Se även: PEEK, POKE, SADD, LEN

Exempel:

```
' FILL ARRAY WITH MACHINE LANGUAGE PROGRAM
DIM CODE%(50)
I = 0
INFOLOOP:
  READ A : IF A = -1 THEN MACHINEPROG:
  CODE%(I) = A: I  = I + 1: GOTO INFOLOOP:
MACHINEPROG:
  X% = 10: Y% = 0
  SETYTOX=VARPTR(CODE%(0))
  CALL SETYTOX(X%,VARPTR(Y%))
  PRINT Y%
END
DATA &H4E56,&H0000,&H206E,&H0008,&H30AE,&H000C,&H4E5E
DATA &H4E75,-1
```

WAVE *kanal, vågdefinition*

Definierar formen på en ljudvåg som skickas till en viss kanal.

Med WAVE kan man exakt definiera hur en ljudvåg, som skickas till högtalarna med SOUND, skall se ut. *vågdefinition* är en indexerad heltalsvariabel som innehåller höjdvärdena (y-värdena) för kurvan som beskriver vågen. Man anger ett höjdnummer i varje element. Höjdnumret är mellan -128 och 127.

vågdefinition kan också vara funktionen SIN.

För att spara minnesutrymme så bör man ta bort variabeln som innehåller vågdefinitionen efter det att WAVE-instruktionen utförts. Det gör man med ERASE-instruktionen.

Exempel:

```
DEFINT A-Z
DIM Timbre(255)
FOR  I=0 TO 255
   READ Timbre(I)
NEXT I
WAVE 0,SIN
WAVE 1,Timbre
WAVE 2,Timbre
WAVE 3,Timbre
```

WHILE *uttryck [satser]* WEND

Utför en serie av satser så länge ett villkor är sant.

Om *uttryck* är sant (dvs resulterar i ett värde skiljt från 0) så utförs alla satser fram till WEND. När WEND påträffas så sker ett hopp tillbaka till WHILE-satsen och *uttryck* räknas ut igen. Om *uttryck* fortfarande är sant så upprepas processen. Om *uttryck* inte är sant så fortsätter programkörningen med satsen efter WEND.

WHILE...WEND kan nästlas hur mycket som helst, det är bara minnesutrymmet som begränsar. Varje WEND matchar den närmast omatchade WHILE-satsen. En omatchad WHILE-sats ger ett "WHILE without WEND" ("WHILE utan WEND") felmeddelande, en omatchad WEND-sats ger ett "WEND without WHILE" ("WEND utan WHILE") felmeddelande.

Varning: Hopp in i WHILE...WEND-slingor förbi WHILE-satsen (dvs den utförs inte) bör undvikas då det kan störa tolkens kontroll av programflödet.

Exempel:

```
' THIS PROGRAM CONVERTS DECIMAL VALUES TO HEXADECIMAL
ANSWER$="Y"
WHILE (ANSWER$="Y")
  INPUT "ENTER DECIMAL NUMBER ", DECIMAL
  PRINT "HEX VALUE OF "  DECIMAL "IS " HEX$(DECIMAL)
  PRINT "OCTAL VALUE OF "  DECIMAL "IS " OCT$(DECIMAL)
  INPUT "DO YOU WANT TO CONVERT ANOTHER NUMBER?  ", ANSWER$
  ANSWER$=UCASE$(ANSWER$)
WEND
END
```

WIDTH *utrustning, [storlek] [,tablängd]*
WIDTH *#filnummer, [storlek] [,tablängd]*
WIDTH *[storlek] [,tablängd]*
WIDTH LPRINT *[storlek] [,tablängd]*

WIDTH-satsen bestämmer antal tecken som kan skrivas ut på en rad och antal tecken mellan tabulatorstoppen för någon valfri utrustning.

utrustning kan vara "SCRN:", "COM1:", eller "LPT1:" och om inget anges så antas "SCRN:".

storlek är ett heltal som anger antal standardtecken som skall rymmas på raden. Vid användning av Amigans proportionella teckenuppsättning så är bredden av ett standardtecken lika med bredden av någon av siffrorna 0 till 9. Om man sätter *storlek* till 255 så blir radbredden "oändlig", dvs tolken kommer inte att framtvinga <RETUR> oavsett hur lång raden blir. Man bör tänka på att POS och LPOS returnerar värdet 0 för pekarens position om den överstiger 255. 255 är förvalt värde för *storlek.*

tablängd är ett heltal som anger antalet standardtecken mellan tabulatorstoppen. Dessa tabstopp avgör sedan var skrivpositionen hamnar när man använder kommatecken tillsammans med PRINT- och LPRINT-satser.

Om man anger "SCRN:" som *utrustning* så definierar man radbredden på skärmen. På grund av att tecknen är proportionella så behöver inte rader med samma antal tecken vara lika långa.

Om man anger "LPT1:" som *utrustning* så definierar man radbredden på skrivaren. WIDTH LPRINT är en alternativ syntax.

När man öppnar filer får de samma radbredd som den utrustning man öppnat den till. Man kan ändra radbredden genom att använda den andra syntaxen som visas ovan.

För mer detaljerad information om den generaliserade I/O-syntaxen, se kapitel 5.

Se även: LPOS, LPRINT, POS, PRINT

WINDOW *fönsterid [,[titel][,[rektangel][,[typ][,skärmid]]]]*
WINDOW CLOSE *fönsterid*
WINDOW OUTPUT *fönsterid*

Skapar resultatfönster, stänger resultatfönster eller gör ett resultatfönster aktuellt utan att göra det till det aktiva fönstret (framflyttat och markerat).

WINDOW-satsen kan göra följande:

* Skapa och visa ett nytt resultatfönster. Fönstret flyttas till positionen längst fram.

* Göra ett fönster aktuellt, dvs man kan använda instruktioner av typen PRINT, CIRCLE och PSET för att skriva text och rita figurer i fönstret.

* Göra ett fönster aktuellt utan att flytta fram det (WINDOW OUTPUT-satsen).

fönsterid är ett nummer mellan 1 och n som identifierar fönstret. Fönstret som öppnas när man startar Amiga Basic är fönster nummer 1, därför skall man numrera alla nya fönster från 2 och uppåt.

titel är ett stränguttryck som visas på fönstrets titelist (om fönstret har en titellist). På fönster 1:s titellist visas namnet på det program som finns i minnet. Om inget program är inläst, så står det "BASIC".

typ bestämmer hur mycket användaren (av ett program) kan påverka ett fönster och om fönstrets innehåll skall sparas då det täcks av andra fönster. Följande värden påverkar *typ*:s funktion:

Värde	Betydelse
1	Fönsterstorleken kan ändras med hjälp av musen och storleksrutan.
2	Fönstret kan flyttas genom att man drar det i titellisten med hjälp av musen.
4	Fönstret kan flyttas i djupled, dvs man kan flytta fönstret så att det kommer bakom eller framför andra fönster. Man använder någon av de två fönsterväxlarna för detta ändamål.
8	Fönstret kan stängas med stängrutan.
16	Fönstrets innehåll bevaras då det täcks över av andra fönster, dvs innehållet kommer att finnas kvar och visas då fönstret inte längre är övertäckt.

typ framställs genom att lägga samman två eller flera av värdena ovan. T ex så anger man 5 som *typ* om man vill att användaren skall kunna flytta fönstret och ändra dess storlek. *typ* får ha ett värde mellan 0 och 31.

Anmärkning: Om man anger 1 och 16 (dvs *typ*=17) så kommer tolken att reservera så mycket minne som krävs för att lagra fönstret när det har sin maximala storlek, dvs minnesåtgången blir mycket stor med denna *typ*.

rektangel anger var fönstret skall placeras på skärmen och fönstrets gränser. *rektangel* anges på formen x1,y1- x2,y2 där x1,y1 är rektangelns övre vänstra hörn och x2,y2 är nedre högra hörnet. Båda koordinaterna är relativa till skärmen. Om man inte anger någon *rektangel* så hamnar fönstret på den plats som *fönsterid*:s senaste värde anger. Det förvalda värdet anger hela skärmen.

skärmid refererar till *skärmid* i en SCREEN-sats. Ange ett värde mellan 1 och 4. Förvalt värde är -1 som identifierar Workbench-skärmen.

WINDOW CLOSE stänger angivet fönster. Om man stänger det aktuella resultatfönstret så kommer det fönster som senast var aktuellt att bli nytt aktuellt resultatfönster.

WINDOW OUTPUT gör att det angivna fönstret blir aktuellt resultatfönster utan att det flyttas till främsta positionen på skärmen. Satser av typen PRINT, CIRCLE och PSET kommer att påverka fönstret. Man kan sålunda ändra i ett fönster utan att det syns.

Exempel:

```
WINDOW 1,"Lines",(10,10)-(270,70),15
WINDOW 2,"Polygons",(310,10)-(580,70),15
WINDOW 3,"Circles",(10,95)-(270,170),15
WINDOW OUTPUT 1
```

Anmärkning: I exemplet ovan kommer fönster nummer 1 ("Lines") att undertrycka Amiga Basic:s resultatfönster.

WINDOW*(n)*

Ger information om det aktuella fönstret.

Program som använder sig av flera fönster behöver information om status och storlek i de olika fönstrena. WINDOW-funktionen ger denna information. Vilken information som returneras beror på värdet av *n:*

n	Returnerad information
0	Det utvalda resultatfönstrets *fönsterid.*
1	Aktuellt resultatfönsters *fönsterid.* Det är detta fönster som PRINT och liknande satser påverkar.
2	Aktuellt resultatfönsters bredd.
3	Aktuellt resultatfönsters höjd.
4	x-koordinaten för den punkt i det aktuella resultatfönstret där nästa tecken ritas.
5	y-koordinaten för den punkt i det aktuella resultatfönstret där nästa tecken ritas.
6	Det maximala antalet färger i det aktuella resultatfönstret.
7	Ger pekaren till aktuellt resultatfönsters INTUITION WINDOW post (se manualen *Intuition:The Amiga User Interface*).
8	Ger pekaren till det aktuella resultatfönstrets RASTPORT post (se manualen *"Intuition: Amiga User Interface").*

WRITE *[uttryckslista]*

Skriver data på skärmen.

Om *uttryckslista* utelämnas så skrivs en tom rad på skärmen, annars så skrivs uttryckens värden. *uttryckslista* kan innehålla både numeriska uttryck och stränguttryck. Uttrycken måste separeras med kommatecken.

När uttrycken skrivs ut så kommer de att separeras av kommatecken; stränguttryck kommer att vara omgivna av citationstecken. När det sista uttrycket är utskrivet så skrivs <RETUR> sist på raden. Till skillnad från PRINT så skriver WRITE ut positiva tal utan inledande blanktecken.

Exempel:

```
A=80 : B=90 : C$="The End"
WRITE A,B,C$
PRINT A,B,C$
END
```

Lägg märke till skillnaden mellan den utskrift som PRINT ger och den som WRITE ger:

```
80,90,"The End"
80             90             The End
```

WRITE# *filnummer, uttryckslista*

Skriver data till en sekventiell fil.

filnummer är det nummer som filen öppnades med i en OPEN-sats. Uttrycken i *uttryckslista* är numeriska uttryck eller stränguttryck. Uttrycken måste separeras med kommatecken.

Skillnaden mellan PRINT# och WRITE# är att den senare automatiskt sätter in kommatecken mellan de poster som skrivs till filen. Man behöver därför inte, som vid PRINT#, ange några speciella åtskiljare i *uttryckslista*. <RETUR> skrivs sist på raden, dvs när alla värden i *uttryckslista* skrivits till filen.

Se även: OPEN, PRINT#, WRITE

Exempel:

```
LET A$ = "32" : LET B = -6 : LET C$ = "Kathleen"
OPEN "O", #1, "INFO"
   WRITE #1,A$;B;C$
CLOSE #1
OPEN "I",#1"INFO"
   INPUT #1,A$,B,C$
   PRINT A$,B,C$
CLOSE #1
END
```

Följande skrivs ut på skärmen:

```
32        -6          Kathleen
```

Appendixs

Appendix A: Teckenkoder

ASCII-teckenkoder

Dec	Hex	Tec	Dec	Hex	Tec	Dec	Hex	Tec
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	‘
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t

Dec	Hex	Tec	Dec	Hex	Tec	Dec	Hex	Tec
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	’	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
041	2AH	*	085	55H	U			

Dec=decimalt, Hex=hexadecimalt (H), Tec=tecken, LF=LineFeed (radframatning), CR=Carriage Return (vagnretur), DEL=Rubout (radering).

Koder för tecken som inte är ASCII-tecken.

b4	b3	b2	b1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
				b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	0	0	0	00			SP	0	@	P	`	p			NBSP	°	À	Đ	à	ð
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç		ç	
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü
1	1	0	1	13			-	=	M	]	m	}			SHY	½	Í	Ý	í	ý
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ
1	1	1	1	15			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

Appendix B: Felkoder och Felmeddelanden

Operationella fel

Felkod Felmeddelande

1 NEXT WITHOUT FOR

NEXT utan FOR. En variabel i en NEXT-sats stämmer inte överens med den närmaste omatchade FOR-variabeln.

2 SYNTAX ERROR

Syntaxfel. Tolken har hittat en programrad som innehåller en otillåten följd av tecken t ex en felstavad instruktion, omatchad parentes etc.

3 RETURN WITHOUT GOSUB

RETURN utan GOSUB. Tolken har hittat en RETURN-sats som inte föregås av någon omatchad GOSUB-sats.

4 OUT OF DATA

Slut på data. En READ-sats har utförts utan att det finns någon DATA-sats med oläst data.

5 ILLEGAL FUNCTION CALL

Felaktigt funktionsanrop. En ogiltig parameter har skickats till en matematisk funktion eller strängfunktion. Man kan även få detta felmeddelande då man använder sig av ett negativt eller för stort index.

6 OVERFLOW

Spill. Resultatet av en uträkning är för stort för att kunna bli representerat i aktuellt format. Om ett resultat blir för litet så sätts resultatet till noll och programkörningen går vidare utan att något fel uppstår.

7 OUT OF MEMORY

Slut på minnet. Ett program är för stort, har för många FOR-slingor eller GOSUB-satser, för många variabler eller för komplicerade uttryck.

8 UNDEFINED LABEL

Odefinierad rad. En angiven rad i en GOTO-, GOSUB-, IF...THEN[...ELSE]-, eller DELETE-sats finns inte.

9 SUBSCRIPT OUT OF RANGE

Index utanför tillåtet område. Felmeddelandet kan framkallas av tre typer av fel:

1. För stort eller för litet index.

2. Fel antal index, dvs fel antal dimensioner.

3. Index används med en variabel som inte är en indexerad variabel.

10 DUPLICATE DEFINITION

Dubbla definitioner. Felmeddelandet framkallas av ett av följande fel:

1. Två DIM-satser för samma indexerade variabel.

2. En DIM-sats för en indexerad variabel som redan antagit det förvalda värdet 10, som övre indexgräns.

3. En OPTION BASE-sats för en indexerad variabel som redan definierats med en DIM-sats eller av det förvalda värdet.

11 DIVISION BY ZERO

Division med noll. Felmeddelandet framkallas av ett av följande fel:

1. En division med noll uppträder i ett uttryck. Resultatet av divisionen blir det största tal datorn kan framställa med samma tecken som täljaren.

2. Noll höjs upp med ett negativt tal. Resultatet av operationen blir det största tal datorn kan framställa.

12 ILLEGAL DIRECT

En instruktion som inte går att använda i interaktivt läge har matats in där. Exempelvis DEF FN går inte att använda i interaktivt läge.

13 TYPE MISMATCH

En strängvariabel tilldelas ett numeriskt värde eller tvärtom. En funktion som skall ha ett numeriskt värde som indata ges ett strängvärde eller tvärtom. SWAP används med en variabel i enkel precision och med en variabel i dubbel precision.

14 OUT OF HEAP SPACE

Slut på systemminnet. Felet kan rättas till genom att tilldela system mer minne med CLEAR- instruktionen, se CLEAR i kapitel 8.

15 STRING TOO LONG

Sträng för lång. Försök att skapa en sträng med fler än 32 767 tecken.

16 STRING FORMULA TOO COMPLEX

Ett stränguttryck är för långt eller för komplext. Felet rättas till genom att bryta ned uttrycket till flera mindre uttryck.

17 CAN'T CONTINUE

Kan ej återuppta programkörning. Ett försök har gjorts att återuppta körningen av ett program som:

1. har stannat på grund av programfel

2. har blivit ändrat efter avbrottet

3. inte finns.

18 UNDEFINED USER FUNCTION

Odefinierad funktion. Ett anrop har skett till en funktion som inte är definierad.

19 NO RESUME

Ingen RESUME-sats finns. Felhanteringsrutin utan RESUME-sats.

20 RESUME WITHOUT ERROR.

RESUME-sats utan fel. En RESUME-sats har utförts trots att inget fel uppstått.

21 UNPRINTABLE ERROR

Inget felmeddelande finns för det uppkomna felet. Detta orsakas oftast av en ERROR-sats med en odefinierad felkod.

22 MISSING OPERAND

Operand saknas. En operator i ett uttryck saknar operand.

23 LINE BUFFER OVERFLOW

Ett försök har gjorts att mata in en rad med för många tecken.

26 FOR WITHOUT NEXT

FOR utan NEXT. FOR-sats utan matchande NEXT-sats.

29 WHILE WITHOUT WEND

WHILE utan WEND. WHILE-sats utan matchande WEND-sats.

30 WEND WITHOUT WHILE

WEND utan WHILE. WEND-sats utan matchande WHILE-sats.

35 UNDEFINED SUBPROGRAM

Odefinierat underprogram. Anrop har skett till ett underprogram som inte finns.

36 SUBPROGRAM ALREADY IN USE

Underprogrammet används redan. Anrop till underprogram som tidigare anropats men inte avslutats. Rekursiva underprogram är inte tillåtna.

37 ARGUMENT COUNT MISMATCH

Antalet parametrar i en CALL-sats är inte detsamma som antalet parametrar i anropad SUB-sats.

38 UNDEFINED ARRAY

En odefinierad indexerad variabel har påträffats i en SHARED-sats.

39-49 UNPRINTABLE ERROR

Inga felmeddelnde finns för dessa felkoder.

Fel i samband med skivhantering

Felkod Felmeddelande

50 FIELD OVERFLOW

En FIELD-sats försöker reservera mer bytes än det som angavs för postlängden i direktåtkomstfilen.

51 INTERNAL ERROR

Internt fel. Ett fel i Amiga Basic-tolken har uppstått.

52 BAD FILE NUMBER

Felaktigt filnummer. En sats eller ett kommando har refererat till ett filnummer som inte är öppnat (med OPEN) eller som är utanför tillåtet talområde.

53 FILE NOT FOUND

Hittar inte filen. Ett FILES-, LOAD-, NAME- eller KILL-kommando eller en OPEN-sats har refererat till en fil som inte finns på

angiven skiva.

54 BAD FILE MODE

Felaktigt filformat. Ett försök har gjorts att:

1. använda PUT, GET eller LOF med en sekventiell fil.

2. läsa in en direktåtkomstfil med LOAD.

3. utföra en OPEN-sats med ett annat filläge än I, O eller R.

55 FILE ALREADY OPEN

Fil redan öppnad. Ett KILL-kommando eller en OPEN-sats med sekventiellt skrivläge specificerat har refererat till en fil som är öppen.

57 DEVICE I/O ERROR

Ett fel uppstod vid I/O till skiva. Felet är fatalt, dvs operativsystemet kan inte ta hand om det.

58 FILE ALREADY EXISTS

Filen finns redan. Det filnamn som specificerats i en NAME-sats finns redan på avsedd skiva.

61 DISK FULL

Skivan full. Allt tillgängligt utrymme på skivan är redan utnyttjat.

62 INPUT PAST END

En INPUT-sats utförs efter det att all data redan lästs från filen eller så försöker man läsa från en tom fil. För att undvika detta fel, använd EOF-funktionen för att testa om filslut nåtts.

63 BAD RECORD NUMBER

Felaktigt postnummer. Postnumret i en PUT- eller GET-sats är antingen för stort eller lika med noll.

64 BAD FILE NAME

Felaktigt filnamn. Ett felaktigt filnamn (t ex ett filnamn med för många tecken) har används som filspecifikation i ett LOAD-, SAVE- eller KILL-kommando, eller i en OPEN-sats.

67 TOO MANY OPENED FILES

För många öppna filer.

68 DEVICE UNAVAILABLE

Utrustning ej tillgänglig. Den utrustning man angivit är inte tillgänglig för tillfället.

70 PERMISSION DENIED (DISK WRITE PROTECTED)

Skivan är skrivskyddad.

73 ADVANCED FEATURE

74 UNKNOWN VOLUME

Okänd skiva. Referens har skett till en skiva som inte finns i angiven skivenhet.

69, 71-73, 75, 76, 78-255 UNPRINTABLE ERROR

Felmeddelande finns inte för dessa felkoder.

Fel som rapporteras innan programkörning påbörjas

SYNTAX ERROR Syntaxfel. Tolken har hittat en programrad som innehåller en otillåten följd av tecken t ex en felstavad instruktion, omatchad parentes mm.

IF without END IF (IF utan END IF)

ELSE/ELSE IF/END IF without IF (ELSE/ELSE IF/END IF utan IF)

BLOCK ELSE/END IF must be the first statement on the line (BLOCK ELSE IF/END IF måste vara första satsen på en programrad)

FOR without NEXT (FOR utan NEXT)

NEXT without FOR (NEXT utan FOR)

WHILE without WEND (WHILE utan WEND)

WEND without WHILE (WEND utan WHILE)

Tried to declare SUB within a SUB
(Försök att deklarera underprogram i underprogram)

SUB alredy defined (Underprogram redan definierat)

Missing STATIC in SUB statement (STATIC fattas i SUB-sats)

EXIT SUB outside of a subprogram
(EXIT SUB-sats utanför underprogram)

SUB without END SUB (SUB utan END SUB)

SHARED outside of a subprogram (SHARED utanför underprogram)

Statement illegal within subprogram
(Satsen otillåten i underprogram)

Too many subprograms (För många underprogram)

Appendix C: Amiga Basic reserverade ord

Nedan visas alla reserverade ord i Amiga Basic. Om dessa ord används som variabelnamn får man ett "Syntax error" felmeddelande.

ABS
ALL
AND
APPEND
AREA
AREAFILL
AS
ASC
ATN

BASE
BEEP
BREAK

CALL
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLNG
CLOSE
CLS
COLLISION
COLOR
COMMON
CONT
COS
CSRLIN
CVD
CVI
CVL
CVS

DATA
DATES
DECLARE
DEF
DEFDBL
DEFINT
DEFLNG
DEFSNG
DEFSTR
DELETE
DIM

EDIT
ELSE
ELSE IF
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXIT
EXP

FIELD
FILES
FIX
FN
FOR
FRE
FUNCTION

GET
GOSUB
GOTO

HEX$

IF
IMP
INKEY$
INPUT
INSTR
INT

KILL

LBOUND
LEFT$
LEN
LET
LIBRARY
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET

MENU
MERGE
MID$
MKD$
MKI$
MKL$
MKS$
MOD
MOUSE

NEW
NEXT
NOT

OBJECT.AX
OBJECT.AY
OBJECT.CLI
OBJECT.CLOSE
OBJECT.HI
OBJECT.OF
OBJECT.ON
OBJECT.PLANES
OBJECT.PRIORITY
OBJECT.SHAPE
OBJECT.START
OBJECT.STOP
OBJECT.VX
OBJECT.VY
OBJECT.X
OBJECT.Y
OCT$
OFF
ON
OPEN
OPTION
OR
OUTPUT

PAINT
PALETTE
PATTERN
PEEK
PEEKL
PEEKW

POINT
POKE
POKEL
POKEW
POS
PRESET
PRINT
PSET
PUT

RANDOMIZE
READ
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN

SADD
SAVE
SAY
SCREEN
SCROLL
SGN
SHARED
SIN
SLEEP
SOUND
SPACE$
SPC
SQR
STATIC
STEP
STICK
STOP
STR$
STRIG
STRING$
SWAP

SUB

TAB
TAN
THEN
TIME
TIMER
TO
TRANSLATE$
TROFF
TRON

UBOUND
UCASE$
USING
USR

VA
VARPTR

WAIT

WAVE
WEND
WHILE
WIDTH
WINDOW
WRITE

XOR

Appendix D: Amiga Basic:s interna representation av tal

Amiga Basic använder binär matematik. Nedan kommer Amiga Basic:s interna representation av tal att visas på hexadecimal form.

Heltal

Heltal representeras med 16-bitars tvåkomplements binära tal med en förteckenbit.

Extern representation	Intern representation
-32768	8000
-1	FFFF
0	0000
1	0001
32767	7FFF

Binär matematik

När binär matematik används så är den förvalda datatypen enkel precision. Inbyggda matematiska funktioner räknar med tal i enkel eller dubbel precision. Uträkningar i enkel precision är mycket snabbare än uträkningar i dubbel precision men det förra är mindre exakt.

Dubbel precision

Använder sig av 8 bytes på följande sätt: 1 förteckenbit följd av 11 förställda exponentbitar och 53 mantissabitar. Om förteckenbiten är 0 så blir talet positivt, är förteckenbiten 1 så blir talet negativt. Den ej förställda exponenten (förställd exponent -3FF hexadecimalt eller -1023 decimalt) är det tal 2 skall upphöjas till innan det multipliceras med mantissan. Mantissan är ett tal större än eller lika med 1 och mindre än 2. Talen som kan representeras är mindre än 1.79*10^308 och större än 2.23*10^-308. Tal i binär dubbel precision representeras med upp till 15.9 siffrors precision.

Extern representation	Intern representation
1	3FF0000000000000
-1	BFF0000000000000
0	000xxxxxxxxxxxxx
10	4024000000000000
0.1	3FB999999999999

Enkel precision

Använder sig av fyra bytes på följande sätt: En förteckenbit följd av 8 förställda exponentbitar och 24 mantissabitar. Om förteckenbiten är 0 så blir talet positivt, är förteckenbiten 1 så blir talet negativt. Den ej förställda exponenten (förställd exponent -7F hexadecimalt, -127 decimalt) är det tal som 2 skall upphöjas till innan det multipliceras med mantissan. Mantissan är ett tal större än eller lika med 1 och mindre än 2. Talen som kan representeras är mindre än 3.4*10^38 och större än 1.18*10^-38. Tal i binär enkel precision representeras med upp till 7.2 siffrors noggranhet.

Extern representation	Intern representation
1	3F800000
-1	BF800000
0	00yxxxxx
10	41200000
0.1	3DCCCCCD

Appendix E: Matematiska funktioner

Nedan visas hur man kan framställa de trigonometriska funktioner som inte finns fördefinierade i Amiga Basic.

Matematisk funktion	**Amiga Basic ekvivalent**
Cotangens	COT(X)=1/TAN(X)
Arcus sinus	ARCSIN(X)=ATN(X/SQR(-X*X+1))
Arcus cosinus	ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
Sinus hyperbolicus	SINH(X)=(EXP(X)-EXP(-X))/2
Cosinus hyperbolicus	COSH(X)=(EXP(X)+EXP(-X))/2
Tangens hyperbolicus	TANH(X)=(EXP(-X)/EXP(X)+EXP(-X))*2+1
Cotangens hyperbolicus	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
Arcus sinus hyperbolicus	ARCSINH(X)=LOG(X+SQR(X*X+1))
Arcus cosinus hyperbolicus	ARCCOSH(X)=LOG(X+SQR(X*X-1)
Arcus tangens hyperbolicus	ARCTANH(X)=LOG((1+X)/(1-X))/2
Arcus cotangens hyperbolicus	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Appendix F: LIBRARY-instruktionens format

Denna information är till för vana programmerare som skall bygga ett bibliotek av maskinspråksrutiner som skall användas från Amiga Basic.

Appendixet beskriver hur Amiga Basic håller reda på rutinnamn och rutiners förskjutning i bibliotekens hopptabell. Amiga Basic måste veta vilka anropsregler som gäller för rutinen ifråga.

Man skapar en speciell fil som innehåller den information som behövs tillsammans med LIBRARY-instruktionen. Om ett bibliotek heter ":Libs/graphics.library" så kommer så kommer den här speciella filen att heta ":Libs/graphics.bmap". Extensionen ".bmap" identifierar filtypen. Formatet på ".bmap"-filen är som följer:

Rutinnamn: n ASCII-tecken, 0-byte avslutar.

Förskjutning i hopptabellen: 16-bitars heltal med en förteckenbit.

Registerparametrar: n bytes på följande sätt:

1=	skicka denna parameter till register d0
2=	skicka denna parameter till register d1
3=	skicka denna parameter till register d2
4=	skicka denna parameter till register d3
5=	skicka denna parameter till register d4
6=	skicka denna parameter till register d5
7=	skicka denna parameter till register d6
8=	skicka denna parameter till register d7
9=	skicka denna parameter till register a0
10=	skicka denna parameter till register a1
11=	skicka denna parameter till register a2
12=	skicka denna parameter till register a3
13=	skicka denna parameter till register a4

0-byte avslutar.

Vid användning av rutiner som följer språket C:s anropsregler och som tar sina parametrar från stacken kommer registerparametern att vara tom då Amiga Basic inte behöver skicka några parametrar till registret.

T ex, om man har ett bibliotek som innehåller dessa två rutiner:

```
MoveTo(x [d0], y[d1]) - library offset = -24 (decimal)

ClearRast(pRast Port [a0] - library offset = -30 (decimal)
```

så skulle en hexadecimal utskrift av bibliotekens ".bmap"-fil se ut så här:

```
4D6F7665546F00FFE8010200436C6561725261737400FFE20900
```

Hjälpprogrammet "ConvertFd.bas" som finns på Extras-skivan framställer en ".bmap"-fil från en ".fd"-fil.

Appendix G: Övningsprogrammet Picture.bas

I detta appendix tar vi oss en närmare titt på programmet Picture.bas som vi använde oss av i kapitel 2. Bokstäverna inom hakparenteser finns inte i programmet på skivan utan är till för din referens.

```
[A] DEFINT P-Z
[B] DIM P(2500)
[C] CLS
[D] LINE (0,0)-(120,120),,BF
[E] ASPECT = .1
[F]  WHILE ASPECT<20
[G]    CIRCLE(60,60),55,0,,,ASPECT
[H]    ASPECT = ASPECT*1.4
[I]  WEND
[J] GET (0,0)-(127,127),P
[K] CheckMouse:
[L]   IF MOUSE(0)=0 THEN CheckMouse
[M]   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
[N]   IF ABS (Y-MOUSE(2)) <3 THEN CheckMouse
[O] MovePicture:
[P]   PUT(X,Y),P
[Q]   X=MOUSE(1): Y=MOUSE(2)
[R]   PUT(X,Y),P
[S]   GOTO CheckMouse
```

I det följande beskrivs rad för rad vad programmet gör.

[A] Variabler vars namn börjar på någon av bokstäverna P till Z definieras till heltalsvariabler.
[B] Definierar en indexerad variabel med 2500 element.
[C] Raderar resultatfönstret.
[D] Ritar och målar en rektangel.
[E] Sätter värdet 0.1 till variabeln ASPECT.
[F] Repeterar alla satser fram till WEND så länge ASPECT är mindre än 20.
[G] Ritar en ellips med centrumpunkten 60,60, radien 55, färg 0 (blå), och ett höjd/bredförhållande lika med ASPECT.
[H] Ökar ASPECT:s värde.
[I] Anger slut på slinga och återhopp till [F] så länge ASPECT<20.
[J] Lagrar angivet område av skärmen i den indexerade variabeln P.
[K] Anger start på subrutinen CheckMouse som kontrollerar musens status.
[L] Väntar på att musens valknapp skall bli nedtryckt.
[M] Om musen flyttas åtminstonde 3 bildelement i x-led så sker ett

hopp till subrutinen CheckMouse (rad [K]).

[N] Om musen flyttas åtminstonde 4 bildelement i y-led så sker ett hopp till subrutinen CheckMouse (rad [K]).

[O] Anger start på subrutinen MovePicture som flyttar bilden lagrad i variabeln P.

[P] Radera bilden på den gamla positionen.

[Q] Sätter musens nya koordinater till variablerna X och Y.

[R] Ritar bilden lagrad i variabeln P på den nya positionen.

[S] Ger hopp till rad [K].

Appendix H: Skapa fonetiska strängar för användning med SAY-instruktionen

Detta appendix beskriver hur man skapar fonetiska strängar som kan omvandlas till ljud i Amigans talsyntesatorn med hjälp av SAY-instruktionen.

Fonem

I ordböcker skriver man med ett fonetiskt alfabet som heter I.P.A (International Phonetic Alphabet). De symboler som används i I.P.A finns dock inte tillgängliga på datorers tangentbord. Man har därför utvecklat ett alfabet som kallas *Arpabet* med vilket man kan representera alla symboler i I.P.A med en eller två versaler. Amigans talsyntesator använder en utvidgad form av Arpabet.

Fonem kan sägas vara talets atomer. Man kan bryta ned en mening i ord, ord i stavelser och stavelser i fonem. Exempelvis så har ordet "cat" tre bokstäver och (av en slump) tre fonem. I fonemtabellen kan man se att tre ljud formar ordet "cat": K, AE och T (KAET). Ordet "cent" representeras av S, EH, N och T (SEHNT). Lägg märke till att trots att de båda orden börjar på "c" så används fonemet K i "cat" medan i "cent" används fonemet "SH". Det finns inget fonem C. Den viktigaste tanken bakom fonetisk skrift är att man skall skriva som det låter - inte som det ser ut.

Välj rätt vokal

I engelskan finns ca 15 vokalfonem, men bara 5 vokalbokstäver: a, e, i, u och o. En och samma bokstav får därför beteckna flera fonem. Se t ex på hur bokstaven "a"står för olika vokalfonem i orden "CAT" och "CAR".

Engelskan har också sk diftonger, dvs vokaler som inte är stationära utan glider från en vokalartikulation till en annan. Jämför den stationära vokalen i "KEEN" och den glidande diftongen i "CANE". Alla vokaler representeras av två symboler enligt nedan:

Vokalfonem

Fonem	Exempel	Fonem	Exempel
IY	beet	IH	bit
EH	bet	AE	bat
AA	hot	AH	under
AO	talk	UH	look
ER	bird	OH	border
AX	about	IX	solid

Diftonger

Fonem	Exempel	Fonem	Exempel
EY	made	AY	hide
OY	boil	AW	power
OW	low	UW	crew

Välj rätt konsonant

Talsyntesatorn skiljer mellan tonande och tonlösa konsonanter. När man uttalar en tonande konsonant vibrerar stämbanden, något som man kan känna genom att lägga fingrarna på struphuvudet. T ex vid "SUDS" är det sista s:et tonande. Ett tonlöst "s" har vi "BATS", dvs här vibrerar inte stämbanden. Kotrollera gärna slutljudande "s", om de är tonande eller inte. Kontrollera också det s k läspljudet. I "THIN" är ljudet tonlöst och stavas fonetiskt THIHN medan det i "THEN" är tonande och återges med DHEHN.

Också "r"-fonemet kräver extra uppmärksamhet. Två fonem finns för "r": R och ER. Om "r" är det enda ljudet i en stavelse så använder man R. Exempel: "absurd" "computer", "flirt". Om "r" följer eller föregår en vokal stavelsen så använder man ER. Exempel: "car", "write", "craft".

Konsonantfonemen återges på följande sätt:

Fonem	Exempel	Fonem	Exempel
R	red	L	yellow
W	away	Y	yellow
M	men	N	men
NX	sing	SH	rush
S	sail	TH	thin
F	fred	ZH	pleasure
Z	has	DH	then
V	very	J	judge
CH	check	/C	loch
/H	hole	P	put
B	but	T	toy
D	dog	G	guest

Förkortningar och specialsymboler

Många fonem i engelska språket skapas av att man förkortar orden när man uttalar dem, man sväljer delar av ordet. T ex i ordet "connector" så försvinner nästan det första o:et. Det betyder att man inte kan använda sig av AA-fonemet utan istället får använda AX. Talsyntesatorn tillåter att man förenklar när man använder sig av dessa vokaler. Istället för att stava "personal" PERSIXNAXL kan man stava det PERSINUL. "Anomaly" kan skrivas UNAAMULIY istället för AXNAAMAXLIY, "combination" skrivs KAAMBINEYSHIN istället för KAAMBIXNEYSHIXN.

Talsyntesatorn använder en del specialsymboler internt och sätter in dem automatiskt i de strängar man skickat till den. Man kan skriva in dessa symboler själv. Q är antagligen den mest användbara specialsymbolen. Q anger att en paus skall ske i uttalet av ordet. I ordet "Atlantic" skall det vara en paus mellan "t" och "l". Talsyntesatorn vet redan att det skall vara en paus där så den kommer att automatiskt att sätta in ett Q. Dock händer det ibland att talsyntesatorn inte sätter in Q på de ställen det behövs i ett ord. Då kan man själv sätta in det där det passar.

Betoning och intonation

För att talsyntesatorn skall producera ett tydligt och naturligt tal behöver den information inte bara om uttalet av enskilda ljud utan även om betoning och intonation. Den behöver veta om meningen är ett påstående eller en fråga, vilket ord eller stavelse som är betonand osv.

För att markera betoning och intonation används ett tecken som består av en siffra 1-9 plus ett vokalfonem. Man placerar siffran efter den vokal som skall betonas och markerar sedan intonationen genom att ange ett visst värde på siffran. Med utgångspunkt i dessa värden räknar sedan talsyntesatorn ut betonings- och intonationskonturer.

Markera betoning

Placera betoningstecknet direkt efter den betonde vokalen.

Exempel:

"CAT" blir KAE5T
"COMMODORE" blir KAA5MAXDOHR
"COMPUTER" blir KUMPYUW5TER

Slå upp ordet i ett lexikon om du är osäker på betoningen.

Om ordet har mer än en betonad stavelse skilj då mellan primär och sekundär betoning genom att ge stavelsen med sekundär betoning ett lägre värde t ex 1 eller 2.

Exempel: "UNDERSTAND" blir AH1NDERSTAE4ND

Skriv ihop sammansatta ord. Den första delen av en sammansättning har ofta en starkare betoning än den andra. Exempel: "LUNCHWAGON" blir LAH5NCHWAE2GIN

Observera att talsyntesatorn ger felmeddelandet bara då tecknet placeras fel; efter en konsonant istället för efter en vokal. Däremot reagerar inte talsyntesatorn om man glömmer att sätta ut betoningstecknet eller placerar det efter en obetonad vokal.

Hur vet jag vilka ord som skall betonas?

Använd ditt eget öra, uttala ordet högt och försök att lyssna; vilket/vilka ord har det starkaste trycket i meningen?

Utgå från ordklasserna. Innehållsord, dvs substantiv, verb och adjektiv har oftast starkt tryck eftersom dessa ord är ämnet för själva texten. Formord som "to", "become", "except" är sällan betonande, de syftar inte på något i verkligheten utan uttrycker bara logiska/grammatiska relationer.

Utgå från meningens informationsstruktur. Ord för nya begrepp, det som talaren tror är nytt för lyssnaren får starkare betoning än det som antas vara känt.

Att ange betoningsgrad

När du nu har placerat ut betoningstecknen på rätt plats är det dags att ange betoningsgraden. Även här kan man utgå ifrån ordklasserna:

substantiv	5	
pronomen	3	
verb	4	
adjektiv	5	
adverb	7	
räkneord	7	
interjektioner	9	(=utropsord "hej", "usch")
artiklar	0	Formord
prepositioner	0	Formord
konjunktioner	0	Formord

Detta är en grov upskattning, som utgår från ordklassernas normala funktion. Ingenting hindrar emellertid att även formord kan få hög betoningsgrad. T ex i följande mening där värdet 9 på ordet "to" visar att brevet skall överlämnas till Mr. Smith personligen: "Please deliver this to Mr. Smith".

Talsyntesatorn känner igen olika skiljetecken och använder dem för att skapa naturliga intonationskonturer. Skiljetecken används på följande sätt:

Punkt - placeras i slutet av alla påstående-meningar, även om meningen bara består av ett ord. Punkt ger en fallton i slutet av meningen.

Frågetecken - skall bara användas i slutet av frågemeningar som kan besvaras med ja eller nej. Med andra ord så skall frågan "Tycker du om din Amiga?" följas av ett frågetecken då den kan besvaras med ja eller nej. Däremot skall frågan "Vilken färg är din favoritfärg?" följas av en punkt då man inte kan besvara den med ja eller nej.

Komma - placeras på vanligt sätt och resulterar i en liten paus och stigande ton, ungefär som om talaren hämtar andan och talar om att han har mer att säga. Använd därför komma för att markera satserna i en mening.

Tankestreck - liknar komma men tonen stiger inte så kraftigt. En tumregel: använd tankstreck runt fraser och komma runt satser.

Parantes placeras dels runt sammansättningar, dels runt längre substantivfraser t ex "the silliest guy I ever saw".

Några tips på vägen

1. Flerstaviga ord är ofta lättare att känna igen än enstaviga. Använd därför hellre "enormous" än "huge".

2. Skriv inte för långa meningar, framförallt inte sådana som kräver andningspauser. Inte mer än en tanke per mening.

INDEX

ABS 8-18
adresser
  format och regler 8-4
adressering av skrivare 5-1
addition av strängar 8-16
adressera extern utrustning 5-1
Amigans snabbvalstangenter 8-2
Amiga Basic minnet 6-11
  använda FRE tillsammans med 6-13
  definera 6-12
  spara 6-12
animering
  accelerera objekt 8-74
  bobs och sprites 7-5
  COLLISION-funktionen 8-31
  definera hastighet 8-79
  definiera objekt 8-77
  definera objekts rörelseutrymme 8-74
  göra objekt synliga eller osynliga 8-75
  definiera kollisionshändelser 8-75
  lokalisera objekt 8-80
  OBJECT-instruktioner 8-74...8-80
  prioritering 8-76
  skapa objekt 7-1
  starta och stanna objekt 8-78
  Se även objekteditorn
AREA 8-18
AREAFILL 8-18
ASC 8-19
ATN 8-20
avhjälpa fel 2-10, 4-4
avbrottshantering 6-8
  BREAK 6-9
  COLLISION 6-9
  MENU 6-9
  MOUSE 6-9
  ON ... GOSUB 6-9
  stoppa 6-10

starta 6-10
tillfälligt avbryta, suspendera 6-10
TIMER 6-9
översikt 6-8
avsluta Amiga Basic 3-2

baudrate, se överföringshastighet
BEEP 8-20
bobs, definera 7-5
BREAK
i avbrottshantering 6-9
instruktionen 8-2
Se även ON BREAK

CALL-instruktionen 8-21
Se även underprogram
CDBL 8-24
CHAIN 8-24
CHDIR 8-26
CHR$ 8-26
CINT 8-27
CIRCLE 8-27
CLEAR
fördela minne med 6-12
instruktionen 8-29
CLNG 8-29
CLOSE 8-29
CLS 8-30
COLLISION
funktionen 8-30
vid avbrottshantering 6-9
objekteditorns förvalde värden 7-1
Se även grafikinstruktioner
COLLISION ON/OFF/STOP 8-31
COLOR 8-32
COM1: 5-1
COMMON 8-32
CONT 8-33
Continue 3-9
Copy 3-8

Copy, snabbval 8-2
COS 8-34
CSNG 8-34
CSRLIN 8-35
Cut 3-8
Cut, snabbval 8-2
CVD 8-35
CVI 8-35
CVL 8-35
CVS 8-35

DATA 8-36
DATE$ 8-37
DECLARE FUNCTION 8-37
DEF FN 8-38
DEFDBL 8-40
DEFINT 8-40
DEFLNG 8-40
DEFSNG 8-40
DEFSTR 8-40
DELETE 8-40
DIM 8-41
direktåtkomstfiler
    exempel 5-11
    läsning av data i 5-10
    skapa 5-9
    översikt 5-8
division
    heltals 8-11
    med noll 8-12

Edit-menyn 3-8
ELSE 8-52
END 8-41
END IF 8-52
END SUB 8-122
Se även underprogram
EOF 8-42
ERASE, instruktionen 8-42

Erase, menyval i objekteditorns Tools-meny 7-5
ERL 8-43
ERR 8-43
ERROR 8-43
EXIT SUB 8-122
EXP 8-44

Felsökning i program 4-4
FIELD 8-45
filnamn 5-2
FILES 8-45
filer
  öppna 5-4
  spara 5-4
FIX 8-46
FOR 8-46
FRE
  användning vid minnesjustering 6-13
  beskrivning 8-47
funktionstangenter, Amigans 8-2
funktionstyper 8-15
färger
  skapa 8-87
  ta reda på färgnummer 8-90
  Se även grafikinstruktioner

GET
  användning tillsammans med direktåtkomstfiler 8-48
  användning vid bildhantering 8-48
  beskrivning 8-48
GOSUB 8-50
GOTO 8-51
grafikinstruktioner
  AREA 8-18
  AREAFILL 8-18
  CIRCLE 8-27
  COLOR 8-32
  LINE 8-60
  PAINT 8-86
  PALETTE 8-87

SCREEN 8-112

heltal
definiera 8-8
korta och långa 8-5
HEX$ 8-51
högupplösning, inställning 8-113
höjd/bredd-förhållande
Amiga skärmens 8-28
definition 8-28
vid cirklar och ellipser 8-28

IF...GOTO 8-52
IF...THEN...ELSE 8-52
indexerade variabler
använda LBOUND och UBOUND 6-7
definera 8-40
delade variabler 6-6
skicka element till underprogram 6-5
variabler som uttryck 6-6
interaktivt arbetsläge 3-3
INKEY$ 8-53
INPUT 8-54
INPUT# 8-56
INPUT$ 8-56
INSTR 8-57
INT 8-57

kommandotangenter, Amigans 8-2
konstanter
decimala 8-5
i dubbel precision 8-6
i enkel precision 8-6
flyttals 8-5
hexadecimala 8-5
korta och långa heltals 8-5
oktala 8-5

KILL 8-58
Klistra, se Paste
konvertering av numeriska typer 8-9

KYBD: 5-1

lågupplösning, inställning 8-113
LBOUND
    beskrivning 8-58
    användning tillsammmans med indexerade variabler 6-7
LEFT$ 8-59
LEN 8-59
LET 8-59
LIBRARY
    beskrivning 8-60
    användning tillsammans med CALL 8-23
LINE 8-60
Line, menyval i objekteditorns Tools-meny 7-5
LINE INPUT 8-61
LINE INPUT# 8-62
LIST, kommandot 8-63
LIST, snabbval 8-2
LIST-fönstret 2-1, 3-6
LLIST 8-63
LOAD 8-63
LOC 8-64
LOCATE 8-64
LOF 8-65
LOG 8-65
LPOS 8-66
LPRINT 8-66
LPRINT USING 8-66
LPT1: 5-1
LSET 8-66
Läsa in ett program 3-2

maskinspråksprogram
    anrop till 8-22
    använda SADD tillsammans med 8-109
    se även underprogram
MENU
    användning vid avbrottshantering 6-9
    beskrivning 8-67

se även underprogram
MENU ON/OFF/STOP 8-68
menyer
Edit 3-8
Project 3-7
Run 3-9
Windows 3-10
menyraden, visa
MERGE 8-68
MID$ 8-69
MKD$ 8-70
MKI$ 8-70
MKL$ 8-70
MKS$ 8-70
MOUSE
beskrivning 8-71
i avbrottshantering 6-9
Se även ON MOUSE
MOUSE ON/OFF/STOP 8-73
musen
position 8-71
status 8-71

NAME 8-73
NEW 8-73
menyval i Amiga Basic:s Project-meny 3-7
menyval i objekteditorns File-meny 7-4
NEXT 8-46, 8-74

objekteditorn
användningsområde 7-1
användningsinstruktion 7-6
menyer 7-2
skärmutseende 7-2
OBJECT.AX 8-74
OBJECT.AY 8-74
OBJECT.CLIP 8-74
OBJECT.CLOSE 8-74
OBJECT.HIT 8-75
OBJECT.OFF 8-75

OBJECT.ON 8-75
OBJECT.PLANES 8-76
OBJECT.PRIORITY 8-76
OBJECT.SHAPE 8-77
OBJECT.START 8-78
OBJECT.STOP 8-78
OBJECT.VY 8-79
OBJECT.VX 8-79
OBJECT.X 8-80
OBJECT.Y 8-80
objekt, hur man skapar dem 7-6, 7-7
OCT$ 8-80
Omvandling av tal se konvertering av numeriska typer
ON BREAK 8-81
ON COLLISION 8-81
ON ERROR 8-82
ON MENU 8-83
ON MOUSE 8-83
ON TIMER 8-84
ON...GOSUB 8-82
OPEN 8-84
Open
- *i Amiga Basic:s Project-meny 3-7*
- *i objekteditorns File-meny 7-4*

operatorer 8-9
- *aritmetiska 8-10*
- *logiska 8-13*
- *relations 8-12*

OPTION BASE 8-86
Oval, i objekteditorns Tools-meny 7-5

PAINT 8-86
Paint, i objekteditorns Tools-meny 7-5
PALETTE 8-87
paritet 5-2
Paste 3-8
Paste, snabbval 8-2
PATTERN 8-88

PEEK 8-89
PEEKL 8-89
PEEKW 8-89
Pen, i objekteditorns Tools-meny 7-5
POINT 8-90
POKE 8-90
POKEL 8-90
POKEW 8-91
POS 8-91
PRESET 8-91
PRINT 8-92
PRINT USING 8-93
PRINT# 8-98
PRINT# USING 8-98
programmeringslägen, olika 3-3
Project-menyn 3-7
PSET 8-100
PTAB 8-101
PUT
  användning tillsammans med direktåtkomstfiler 8-102
  användning vid bildhantering 8-102
  beskrivning 8-101

Quit, i objekteditorns File-meny 7-4

RANDOMIZE 8-103
READ 8-103
Rectangle, i objekteditorns Tools-meny 7-5
redigera program 4-1, 2-7
REM 8-104
RESTORE 8-104
RESUME 8-105
Resultatfönstret 3-5, 2-2
RETURN 8-105
RIGHT$ 8-106
RND 8-107
RSET 8-107
rulla programlistning 4-3

RUN, kommandot 8-108
Run-menyn 3-9

SADD 8-109
SAVE 8-109
Save
- i Amiga Basic:s Project-meny 3-7
- i Objekteditorns File-meny 7-4

Save As
- i Amiga Basic:s Project-meny 3-7
- i objekteditorns File-meny 7-4

spara ett program 3-3, 2-17
SAY 8-110
SCREEN
- använda systemminnet tillsammans med 6-12
- beskrivning 8-112

SCREEN CLOSE 8-112
skärmlägen, hur man sätter olika 8-112
SCRN: 5-1
SCROLL 8-113
sekventiella filer
- lägga till data 5-7
- läsa data från 5-7
- skapa 5-6
- översikt 5-5

SGN 8-114
SHARED 8-114
Show List, menyval i Amiga Basic:s Windows-meny 3-10
Show Output, menyval i Amiga Basic:s Windows-meny 3-10
SIN 8-115
SLEEP 8-116
slingor, nästlade 8-47
SOUND
- använda systemminnet tillsammans med 6-12
- beskrivning 8-116

SPACE$ 8-118
SPC 8-119
specialtecken 8-1
spill 8-12

sprites, definiera sprites med objekteditorn 7-6
SQR 8-119
stacken
- *använda FRE med 6-13*
- *definition 6-11*
- *spara utrymme i 6-11*

Start, menyval 3-9
Start, snabbval 8-2
starta Amiga Basic 3-1
STATIC 8-122
Step, vid felsökning 4-5, 3-10
STICK 8-120
Stop, menyval 3-9
STOP 8-121
STR$ 8-121
STRING$ 8-122
Strängar 8-5, 8-8, 8-15
SUB 8-122
se även underprogram
Suspend, menyval 3-9
SWAP 8-124
SYSTEM 8-124
systemminnet
- *använda FRE tillsammans med 6-13*
- *definition 6-12*
- *spara utrymme 6-12*

TAB 8-125
tal, skapa syntetiskt
- *manuellt Appendix H*
- *med SAY 8-110*
- *med TRANSLATE$ 8-127*

taltyper, uppräkning av Amiga Basic:s olika taltyper 8-5
TAN 8-125
THEN 8-52
TIME$ 8-126
TIMER, instruktionen 8-127
TIMER, vid avbrottshantering 6-9
TIMER ON/OFF/STOP 8-126

Trace Off, menyvalet 3-9
Trace On, menyvalet 3-9
TRANSLATE$ 8-127
TROFF 8-127
TRON
- *beskrivning 8-127*
- *vid felsökning 4-4*

underprogram
- *anropa 8-21*
- *avgränsa 6-3*
- *delade variabler i 6-6*
- *fördelar 6-1*
- *referera i CALL 8-21, 6-1*
- *referera indexerade variabler i 6-5*
- *statiska variabler i 6-4*

upplösning, skärmens 8-113
uttryck 8-9

VAL 8-129
variabler 8-6
- *definera 8-7*
- *indexerade 8-8*

VARPTR 8-129
välja List-fönstret 3-6
välja text 4-2

WAVE
- *använda systemminnet tillsammans med 6-12*
- *beskrivning 8-130*

WEND 8-131
WHILE...WEND 8-131
WIDTH 8-132
WINDOW
- *använda systemminnet tillsammans med 6-12*
- *funktionen 8-135*
- *instruktionen 8-133*

fönster 8-133
- *skapa 8-133*
- *få information om 8-135*

Windows-menyn 3-10
WRITE 8-136
WRITE# 8-137

överföringshastighet 5-2

Commodore

This was brought to you

from the archives of

http://retro-commodore.eu